İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

SECURE ANTS: A KEYNOTE IMPLEMENTATION FOR ACTIVE NODE TRANSFER SYSTEM

M.Sc. Thesis by

Tarık ÇINAR

(504981185)

Date of submission : 8 June 2001

Date of defence examination: 28 June 2001

Supervisor (Chairman): Prof. Dr. Bülent ÖRENCİK

Members of the Examining Committee Assoc. Prof.Dr. Sema OKTUĞ (İ.T.Ü.)

Assoc. Prof.Dr. Haluk GÜMÜŞKAYA (M.Ü.)

JUNE 2001

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

GÜVENLİ ANTS: ACTIVE NODE TRANSFER SYSTEM İÇİN KEYNOTE UYGULAMASI

YÜKSEK LİSANS TEZİ
Tarık ÇINAR
(504981185)

Tezin Enstitüye Verildiği Tarih : 8 Haziran 2001
Tezin Savunulduğu Tarih : 28 Haziran 2001

Tez Danışmanı : Prof.Dr. Bülent ÖRENCİK
Diğer Jüri Üyeleri Doç.Dr. Sema OKTUĞ (İ.T.Ü.)
Doç.Dr. Haluk GÜMÜŞKAYA (M.Ü.)

HAZİRAN 2001

ACKNOWLEDGMENTS

I thank Bülent ÖRENCİK for being a great thesis advisor. I thank him for his good advice on how to write and present a research. I also thank my family whose continual love and support throughout my whole life have kept me going.

Tarık ÇINAR

June 2001

CONTENTS

LIST OF TABLES

LIST OF FIGURES

SECURE ANTS: A KEYNOTE IMPLEMENTATION FOR ACTIVE NODE TRANSFER SYSTEM

SUMMARY

The current network infrastructure is essentially static. Although active code may be sent from servers to clients (such as web applets) and from clients to servers (such as OO database queries), internal network nodes (such as routers) passively switch packets. This infrastructure is standardized using monolithic protocols such as IP. Adding functionality to these core network protocols is performed by adding complexity to the protocols through a lengthy process. The result is that although the core protocols become bloated, they are still incapable of incorporating all of the functionality within the network that applications (such as convergecasts or data caching) leverage network-based computation and storage.

The desire for flexible networking services has given rise to the concept of "active networks". Active networks provide a general framework for designing and implementing network-embedded services, typically by means of a programmable network infrastructure. Active networks allow their users to inject customized programs into the nodes of the network. Packets are replaced with "capsules", program fragments that are executed at each active node they traverse. A programmable network infrastructure creates significant new challenges for securing the network infrastructure. Traditional authorization mechanisms are inadequate for handling security of active networks. The "trust-management" approach, "PolicyMaker" and "KeyNote" trust-management engines were developed as an answer to the inadequacy of these systems. Trust-management engines avoid the need to resolve "identities" in an authorization decision. Instead, they express privileges and restrictions in a programming language. This allows for increased flexibility and expressibility, as well as standardization of modern, scalable security mechanisms. Using these privileges and restrictions provides security of active applications runs on an active node and it is the main advantage of these engines for active networks.

In this thesis, we examine Active Node Transfer System (ANTS) as an active network, trust-management approach, PolicyMaker and KeyNote trust-management engines and we implement KeyNote trust-management engine in Java language for ANTS.

GÜVENLİ ANTS: ACTIVE NODE TRANFER SYSTEM İÇİN KEYNOTE UYGULAMASI

ÖZET

Günümüz ağ altyapısı temel olarak statiktir. Aktif kod, sunuculardan istemcilere java applet'leri gibi ve istemcilerden sunuculara veri tabanı sorgulamaları gibi gönderilebilmekle beraber, dahili ağ düğümleri paketleri pasif olarak anahtarlamaktadır. Bu altyapı, IP gibi monolitik protokoller kullanılarak standartlaştırılmıştır. Bu temel ağ protokollerine yeni uygulamalar ekleme, uzun bir süreci içermekte, bu protokolleri karmaşık hale getirmekte, çoklu aktarım ve bilgi kaynaşımı gibi ağ tabanlı hesaplama ve bellek gereksinimi olan uygulamaları içermemektedir.

Bu tip esnek ağ uygulamalarının gerçeklenmesi isteği "aktif ağlar" kavramını ortaya çıkarmıştır. Aktif ağlar, programlanabilir bir ağ altyapısı sunarak, kullanıcılarının ağdaki aktif düğümlere özelleştirilmiş programları yükleyebilmesine imkan tanımaktadır. Paketler, "kapsül" olarak nitelenen ve geçtikleri aktif düğümler üzerinde yürütülen program parçaları ile değiştirilmiştir. Bununla birlikte aktif ağların programlanabilir olması, ağ altyapısının güvenliğinin sağlanması açısından önemli tehditler ortaya çıkarmıştır. Geleneksel yetkilendirme mekanizmaları aktif ağların güvenliğinin sağlanmasında yeterli olmamaktadır. Bu eksikliği gidermek amacıyla "güven yönetimi" kavramı ve bu kavramı temel alan "PolicyMaker" ve "KeyNote" sistemleri geliştirilmiştir. Bu sistemler bir yetkilendirme işleminde, kimlikleri tanımlamak yerine bir programlama dilinde öncelik ve kısıtlamaları tanımlamaktadır. Bu tanımlama, esnekliği ve açıklığı artırmakta, modern ve ölçeklenebilir güvenlik mekanizmalarının oluşturulmasını kolaylaştırmaktadır. Aktif ağlar düşünüldüğünde bu sistemlerin en büyük avantajı ise aktif düğümler üzerinde yürütülen aktif uygulamaların güvenliğinin, bu uygulamalar için tanımlanmış öncelik ve kısıtlamalarla sağlanabiliyor olmasıdır.

Bu çalışmada, bir aktif ağ olan Active Node Transfer System(ANTS), güven yönetimi kavramı, PolicyMaker ve KeyNote sistemleri incelenmiş ve KeyNote sistemi java programlama dili ile yazılarak, ANTS üzerinde uygulanmıştır.

CHAPTER 1

INTRODUCTION

Traditional data networks passively transport bits from one end system to another. Ideally the user data is transferred opaquely, i.e., the network is insensitive to the bits it carries and they are transferred between end systems without modification. The role of computation within such networks is extremely limited, e.g., header processing in packet-switched networks and signalling in connection-oriented networks.

Active networks [1,2] break with tradition by allowing the network to perform customized computations on the user data. For example, a user of an active network could send a customized compression program to a node within the network (e.g., a router) and request that the node execute that program when processing their packets. These networks are "active" in two ways. First, switches perform computations on the user data flowing through them. Second, individuals can inject programs into the network.

1.1 Threats

Threats to network infrastructure are intimately tied to the model used for sharing the infrastructure. For example, with the unreliable best-effort model provided by the Internet and lack of per-hop security properties, security policies are enforced end-to-end.

IP [3] packets are anonymous to the routers, and they, at least before extensions such as multicasting (e.g., MBONE [4]) and RSVP [5], are allocated service on a FIFO basis. IPSEC [6] provides authentication services, but it remains unclear how support

for Quality of Service (such as RSVP) will be integrated with authentication services. As it stands, the Internet infrastructure is vulnerable to a variety of denial of service attacks as a consequence of minimal resource accountability, as well as a variety of other attacks such as traffic analysis. We note that since the resource model in the routers is so simple, sophisticated threats are posed by attacks on services implemented at the endpoints, e.g., the notorious "Syn-Ack" (also known as "Syn-flooding") attack [7] on TCP/IP [3].

Active Networks, being more flexible, considerably expands the threat possibilities. The security threats faced by such elements are considerable. For example, when a packet containing code to execute arrives, the system typically must:

- Identify the sending network element,
- Identify the sending user,
- Authorize access to appropriate resources based on these identifications,
- Allow execution based on the authorizations and security policy.

The principals involved in the authorization and policy decisions in the security model are users, programmers and administrators and network elements. The network elements are presumed to be under physical control of an administrator. Programmers may not have physical access to the network element, but may possess considerable access rights to resources present in the network elements. Users may have access to basic services (e.g., transport), but only resources that the network elements are willing to export to all users, at an appropriate level of abstraction. Users may also be allowed to introduce their own services, or load those written by others.

In networking terminology, the first three steps comprise a form of admission control, while the final step is a form of policing. A second separation is that of static versus dynamic checking. Security violations occur when a policy is violated, e.g., reading a private packet, or exceeding some specified resource usage.

1.2 Background

Existing authorization mechanisms fail to provide powerful and robust tools for handling security at the scale necessary for today's Internet. These mechanisms are coming under increasing strain from the development and deployment of systems that increase the programmability of the Internet. Moreover, this "increased flexibility through programmability" trend seems to be accelerating with the advent of proposals such as Active Networking [1].

The trust-management approach [8] to distributed-system security was developed as an answer to the inadequacy of traditional authorization mechanisms. Trust-management engines avoid the need to resolve "identities" in an authorization decision. Instead, they express privileges and restrictions in a programming language. This allows for increased flexibility and expressibility, as well as standardization of modern, scalable security mechanisms. Further advantages of the trust-management approach include proofs that requested transactions comply with local policies and system architectures that encourage developers and administrators to consider an application's security policy carefully and specify it explicitly.

1.2.1 Trust Management

A traditional "system-security approach" to the processing of a signed request for action treats the task as a combination of authentication and access control. If we consider the traditional capability systems as ACLs, the receiving system first determines who signed the request and then queries an internal database to decide whether the signer should be granted access to the resources needed to perform the requested action. In a large, heterogeneous, distributed system, there is a huge set of people (and other entities) who may make requests, as well as a huge set of requests that may be made. These sets change often and cannot be known in advance. Even if the question "who signed this request?" could be answered reliably, it would not help in deciding whether or not to take the requested action if the requester is someone or something from whom the recipient is hearing for the first time.

The right question in a far-flung, rapidly changing network becomes "is the key that signed this request authorized to take this action?" Because name-key mappings and pre-computed access-control matrices are inadequate, one needs a more flexible,

more "distributed" approach to authorization. The trust-management approach [8] frames the question as follows: "Does the set C of credentials prove that the request r complies with the local security policy P?" Each entity that receives requests must have a policy that serves as the ultimate source of authority in the local environment. The policy may directly authorize certain keys to take certain actions, but more typically it will delegate this responsibility to credential issuers that it trusts to have the required domain expertise as well as relationships with potential requesters. The trust-management engine is a separate system component that takes (r; C; P) as input, outputs a decision about whether compliance with policy has been proven, and may also output some additional information about how to proceed if it hasn't.

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

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

- How should "proof of compliance" be defined?
- Should policies and credentials be fully or only partially programmable? In which language or notation should they be expressed?
- How should responsibility be divided between the trust-management engine and the calling application? For example, which of these two components should

perform the cryptographic signature verification? Should the application fetch all credentials needed for the compliance proof before the trust-management engine is invoked, or may the trust-management engine fetch additional credentials while it is constructing a proof?

1.2.2 PolicyMaker

PolicyMaker [8,9] 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 principle. It addressed the authorization problem directly, rather than handling the problem indirectly via authentication and access control, and it provided an application-independent definition of "proof of compliance" for matching up requests, credentials, and policies.

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

PolicyMaker assertions can be written in any programming language that can be "safely" interpreted by a local environment that has to import credentials from diverse (and possibly untrusted) issuing authorities. 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.

1.2.3 KeyNote

KeyNote [10] was designed according to the same principles as Policy-Maker, using credentials that directly authorize actions instead of dividing the authorization task into authentication and access control. Two additional design goals for KeyNote were standardization and ease of integration into applications. To address these goals, KeyNote assigns more responsibility to the trust-management engine than PolicyMaker does and less to the calling application; for example, cryptographic signature verification is done by the trust-management engine in KeyNote and by the application in PolicyMaker. KeyNote also requires that credentials and policies be written in a specific assertion language, designed to work smoothly with KeyNote's compliance checker. By fixing a specific and appropriate assertion language, KeyNote goes further than PolicyMaker toward facilitating efficiency, interoperability, and widespread use of carefully written credentials and policies.

A calling application passes to a KeyNote evaluator a list of credentials, policies, and requester public keys, and an "Action Environment". This last element consists of a list of attribute/value pairs, similar in some ways to the Unix TM shell environment. The action environment is constructed by the calling application and contains all information deemed relevant to the request and necessary for the trust decision. The action-environment attributes and the assignment of their values must reflect the security requirements of the application accurately. Identifying the attributes to be included in the action environment is perhaps the most important task in integrating KeyNote into new applications. The result of the evaluation is an application-defined string (perhaps with some additional information) that is passed back to the application. In the simplest case, the result is something like "authorized". As in PolicyMaker, policies and credentials (collectively called assertions) have the same format. The only difference between policies and credentials is that a policy (that is, an assertion with the keyword POLICY in the Authorizer field) is locally trusted (by the compliance-checker) and thus needs no signature.

In PolicyMaker, compliance proofs are constructed via repeated evaluation of assertions, along with an arbitrated "blackboard" for storage of intermediate results and inter-assertion communication. In contrast, KeyNote uses a depth-first search (DFS) algorithm that attempts (recursively) to satisfy at least one policy assertion.

1.3 Goals

The Active Network Transfer System (ANTS) [11] was one of the first active packet systems developed. The current ANTS prototype is written in Java [12] and relies upon the JVM's bytecode verification and sandboxing facilities [13] for the safety features they provide. Local node resource usage is governed through the use of watchdog timers and memory allocation limits. Capsule types are grouped into protocols, and capsules are restricted to only access soft state belonging to their own protocol. The reference to the code actually takes the form of a MD5 [14] cryptographic hash of the actual code, thus preventing code spoofing. Thus, a misbehaving capsule is isolated from other capsules and the node itself, and if it consumes too many resources it is terminated. To control network-wide resource use, ANTS provides a TTL field, which is decremented at each hop, and duplicated when packets create a child packet. Since packets can create any number of child packets, the TTL limits the distance a packet's children can travel, but not the total work in the network.

ANTS does not provide any type of authentication or traditional security schemes to ensure the safety of running foreign code on an active node [11]. Instead, it relies on the safety mechanisms of Java [12] (e.g., byte-code verification) to execute untrusted code. This, of course, is not sufficient to ensure a robust network. It is clear that additional security mechanisms are necessary to ensure secrecy, integrity, and availability for all the users in the network. The goal of this thesis is to solve authentication problem of ANTS by verifying packet payload and to solve authorization problem of ANTS by implementing KeyNote [10] trust-management engine.

1.4 Thesis Overview

In the next chapter, we examine PGP [15] and X.509 [16,17] as traditional Public Key Infrastructure (PKI) and QCM [18-20] as an active network PKI. In Chapter 3, we present an overview of active networks and ANTS [11], a toolkit for building active network applications. In Chapter 4, we present our design and implementation of KeyNote [10] trust-management engine. Finally, Chapter 5 draws conclusions of

our solution, discusses some issues that need further investigation, and suggests some possible future work.

CHAPTER 2

RELATED WORK

This chapter gives information about traditional and Active Network PKI. Section 2.1 describes PGP PKI. Section 2.2 describes X.509 PKI. Finally Section 2.3 describes QCM PKI.

2.1 PGP

In the PGP system [15], a user generates a (PublicKey; SecretKey) pair that is associated with his unique ID; usually an ID is of the form (Name;EmailAddress). Keys are stored in key records. A public (resp. secret) key record contains an ID, a public (resp. secret) key, and a timestamp of when the key pair was created. Public keys are stored on public key rings and secret keys on secret key rings. Each user must store and manage a pair of key rings.

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

record to judge the key it contains valid, and a less skeptical user may require only one fully trusted signature or two marginally trusted ones.

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

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

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

2.2 X.509

The X.509 [16,17] authentication framework attempts to solve the same part of the trust management problem that PGP's introducer mechanism attempts to solve, namely the need to find a suitably trustworthy copy of the public key of someone with whom one wants to communicate. As in PGP, X.509 certificates are signed records that associate users' IDs with their cryptographic keys; X.509 certificates contain more information than PGP certificates, e.g., the names of the signature schemes used to create them and the time interval in which they are valid, but their basic purpose is simply the binding of users to keys. However, X.509 differs sharply from PGP in its level of centralization of information. While anyone may sign

public-key records and act as an introducer in PGP, the X.509 framework postulates that everyone will obtain certificates from an official certifying authority (CA). When user A creates a (PublicKey; SecretKey) pair, he has it and the rest of the required information certified by one or more CAs and registers the resulting certificates with an official directory service. If A later wants to communicate securely with B, he obtains a certificate for B from one of the directory servers. If A and B have both been certified by the same CA, the directory server can just send B's certificate to A, who can verify its validity using the public key of this common CA. If A and B have not been directly certified by a common CA, then the directory service must create a certification path from A to B. This is a list of the form CA_1, $cert_1$, CA_2, $cert_2$,..., CA_n, $cert_n$, where $cert_i$, $1 \leq i < n$, is a certificate of CA_{i+1} that has been signed by CA_i, and $cert_n$ is a certificate of B. In order to use this path to obtain B's public key, A must know the public key of CA_1, the first authority in the path. Thus, the X.509 framework rests on the assumption that CAs are organized into a global "certifying authority tree" and that all users within a "community of interest" have keys that have been signed by CAs with a common ancestor in this global tree.

2.3 QCM

QCM [18-20] stands for "Query Certificate Manager"; it is a software system that has been developed at the University of Pennsylvania as part of the SwitchWare project on active networks. QCM is a Public Key Infrastructure (PKI) intended to support secure maintenance of distributed data sets like Access Control Lists (ACL's) or public key certificate repositories. An ACL is a list of "principals", identified by public keys; such lists can be used to describe who is permitted to access resources such as the ability to read and modify a file, or run a program. A public key certificate is an association between a public key and an individual or entity. QCM allows policies, such as the ACL of principals allowed to access a resource, to be described in a special-purpose language (the language is also called QCM). The system provides two services. First, it verifies whether a policy is satisfied by a request, and, second, it uses the policy verification to assist in retrieving the certificates (digitally signed documents) that are relevant to the verification. This integrated verification and retrieval mechanism is known as policy directed certificate retrieval and is the primary novel contribution of the QCM system.

Two QCM-aware applications have been built so far. The first application [21] provides policies for the evaluation of PLAN programs. PLAN [22] is a programming language for active networks, that is, it is a language for programming the network routing elements, which forward packets in an internet. Internets that support such programmability are called active networks. PLAN allows QCM to provide access control policies for functions on active routers invoked by PLAN packets. This capability has been applied [23] to the development of an active network firewall, that is, an active network router that examines packets to provide security for a portion of a network. The second application [24] provides ACL maintenance for a test bed of computers for active networks known as the ABONE [25]. The ABONE uses a program called ANET to allow users to set up active network experiments on a collection of machines located around the world. The testbed allows active networks to be tested in the context of actual Internet traffic, but significant security concerns are raised by availability on the public network, which is notorious for mischief-makers. QCM provides support for ANET's ACL's, which determines who is allowed to use the ABONE.

CHAPTER 3

ACTIVE NETWORKS

Active networks [26] provide a new way of thinking about a network environment. Traditional networks are based on an "end-to-end" model, in which computation is done at the endpoints, with only routing and header processing occurring within the network. All packets are routed through a network independent of content. Active networks break from this model by allowing computation to occur within the network. In an active network, the nodes (i.e., the routers and switches) can perform computations on individual packets in addition to routing them. This extra computation can be defined by users, who can inject programs into the network to customize processing of user and/or application specific data. This chapter discusses active networks in general and a prototype implementation of an active network called ANTS [11].

3.1 Architecture

There are two approaches to building an active network, a discrete and an integrated one [26]. In the following sections, we present both approaches with more emphasis on the integrated approach used by ANTS.

3.1.1 A Discrete Approach

The discrete approach separates the mechanism for injecting programs into a "programmable" node from the actual processing of packets as they flow through a node. Users send a program to a node as they would to a host. This program would then be stored at the node. When a packet arrives at the node, the corresponding program is selected using some header information and then executed. When a new

version of the program is necessary, or if a different type of processing is required, the user can send the new program to the node to replace the old one.

This approach is definitely preferable when programs are relatively large compared to the packets. This approach also maintains a modularization between user data and program, which may be useful for network management tasks.

3.1.2 An Integrated Approach

A different approach, the one used by ANTS, is to integrate the program with every packet. In this manner, every packet that is sent contains not only data, but also a program. The term capsule is used to describe these new types of packets. When a capsule arrives at a node, its contents are evaluated using the program in the capsule.

The traditional router or switch, which is responsible for routing and header processing, is replaced by an active node. In addition to capsule routing, an active node consists of three major components: a code loading mechanism, a transient execution environment, and a more permanent storage area. When a capsule arrives at an active node, the code associated with the capsule is loaded by the code loading mechanism. The capsule and the program loaded are then passed on to a transient execution environment. There the program is allowed limited access to node resources such as memory and other storage components. There is a restricted set of capsule primitives that comprise all the actions that a capsule can perform at a node. Capsules are also allowed to retrieve or store information in a more permanent storage area, the node cache. The result of processing may involve zero or more capsules being sent out into the network and/or a modification of the node cache. When processing is completed, the transient environment is destroyed and resources held by the capsule are released.

3.2 ANTS - An Active Network and Toolkit

ANTS is an active network toolkit developed at the Massachusetts Institute of Technology (MIT) Laboratory for Computer Science (LCS) by D. J. Wetherall [11]. ANTS is toolkit to help build and maintain active network applications. ANTS also includes an implementation of an integrated active network. ANTS is written in Java and runs as a user-level application on the Linux operating system.

The entire ANTS environment consists of three major classes, as shown in Table 3.1. The Application class is the user interface part of the environment. It is also responsible for sending and receiving capsules, which are implemented by the Capsule class. The Node class is the implementation of an active node.

Table 3.1 Key Classes and Methods [11]

Class	Methods
Application	send, receive (upcall), node
Capsule	evaluate, length, register, serialize, deserialize
Node	address, get, put, routefornode, delivertoapp

3.2.1 Applications

Active network applications are written by specializing the Application class. This class provides some basic methods that interact with the node it runs on to send and receive capsules. Before an application can send a capsule, however, it must first register the capsule type with the local node using the Capsule.register method. This is a requirement imposed by the code distribution protocol [see Section 3.2.3]. Once the capsule type is registered, an application can send instances of the capsule type into the active network environment using the send method. In addition, an application must have a receive method, which will be executed when a capsule arrives at the node. Currently, only one application can be running on a node. In the future, multiple applications can run on a node, and incoming capsules will be directed to their respective applications.

3.2.2 Capsules

In ANTS, capsules are implemented by the Capsule class. Capsules have a specific format as shown in Figure 3.1. The first 8 bytes of a capsule is the capsule ID, which is used to identify the capsule type and where the code for the capsule resides. It is represented in the Capsule class as the MethodID (MID), which must be unique for

each capsule type1. The MID is divided into three parts, an authority (4 bytes), a group (2 bytes), and a subgroup or member (2 bytes). As each capsule type may have unique and customized routines, there must be a way to locate the code and transport it to a particular node when necessary. The authority is the identifier of the node that is the repository of the processing routines associated with this capsule type. The group and member value provide a shallow hierarchy for capsules. If two capsules are of the same group, they are considered as part of a single protocol, and their associated routines are transported around the network as a single unit. This is useful for applications that spawn capsules in the middle of the network.

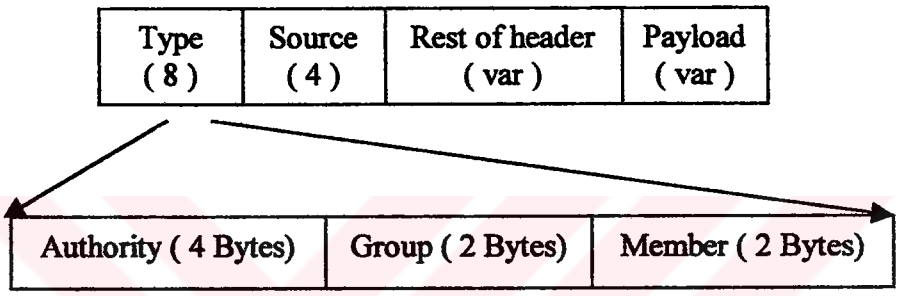

Figure 3.1 Capsule Format [11]

The next 4 bytes indicate the source address of the capsule; currently, the system uses IPv4 style addresses [27]. The rest of the capsule contains a variable header and payload, depending on the type of capsule. The variable header field contains, among other things, information necessary for routing, e.g., the destination address. The payload field may contain data that is necessary for capsule processing at a node.

There are actually two different types of capsules: user capsules and system capsules. Normal applications use user capsules. A system capsule has more privileges and is used by ANTS itself for special purposes. For example, DLRequestCapsule and DLResponseCapsule are system capsules used by the code distribution mechanism. The capsule class hierarchy is shown in Figure 3.2.

Each capsule type must also provide a certain set of required methods. For marshaling purposes, the developer must provide a length method which returns the

1 Currently, ANTS does not provide a mechanism to generate unique MethodIDs. Users must generate their own MethodID and hope there is no collision within the network. A future version will use a server hash of the code as the MethodID.

length of the capsule in bytes. In addition, a serialize and deserialize method are required to marshal objects into data streams for transmission and back. For capsule processing, an evaluate method is required. This is the heart of the capsule; this method dictates all the processing that will be done when a capsule arrives at a node. Finally, before a capsule can be used (sent into a network) by an application, it must first be registered. Registering a capsule indicates that the capsule routine resides at a particular node. This is necessary for the dynamic code loading mechanism explained in the next section.

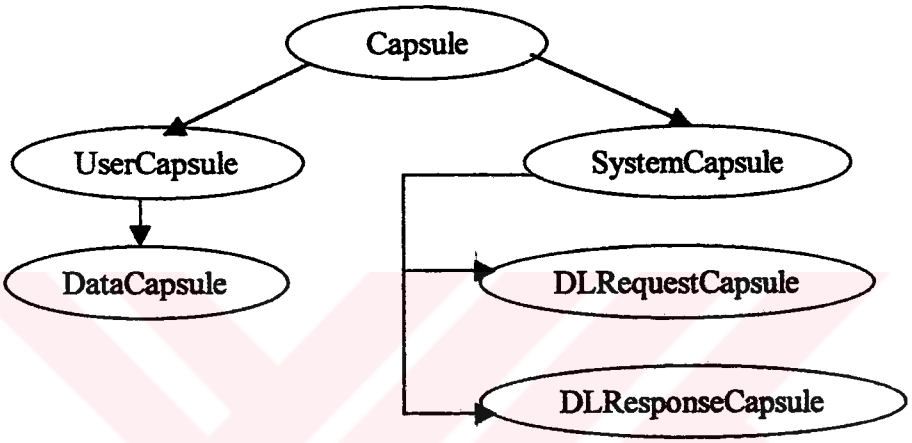

Figure 3.2 Capsule Class Hierarchy

3.2.3 Active Nodes

Active nodes are the processing engines that receive capsules, load capsule routines, process the routines, and schedule capsules for further transmission if necessary. They are basically the environment in which capsules run. An active node, including the runtime, object cache, and demand loading protocol, is implemented by the Node class. This class provides a set of "primitives" used by capsules. The address method returns the address of the current node. The get and put methods give capsules the ability to store and retrieve objects in the node cache. The cache stores objects for a period of time, as requested by capsules, but not exceeding a maximum time dictated by the node itself. The cache uses a least-recently-used (LRU) strategy to evict items. The node offers no guarantee that an object is cached for the duration of time specified by the capsule or for the maximum cache time since the object may be evicted before it times out. The routefornode method is used to route packets to their destination node. If the current node is the destination node, the delivertoapp method is used to deliver the capsule to the application.

The demand loading protocol, the mechanism that loads capsule programs into a node, works as shown in Figure 3.3. When a capsule arrives at a node (1), the node first checks its cache to see if the code is already present. If so, no further code loading is necessary. If not, the node will query the last node visited by the capsule for the code (2). If the previous node has the code, it will respond, and the code will be loaded (3). It is unlikely that a node cannot retrieve the necessary code from the previous node because either (a) the previous node just processed the capsule before routing it to the current node, which means the code is in the node cache of the previous node, or (b) the previous node is the originator of the capsule, which means an application running on that node must have registered the capsule [see Sections 3.2.1 and 3.2.2]. When an application registers a capsule, it indicates that it has the processing routines associated with the capsule or, at least, knows how to find them. Hence, the node that injected the capsule into the network will always be able to produce the capsule routines. On the other hand, it is possible for a previous node (not the originator) that just processed the capsule to not be able to produce the code.

Due to a great number of capsules going through that node at a particular time, the capsule routines may have been flushed from the cache before the request arrived. If this is the case, the querying node will use the authority field in the MethodID of the capsule to retrieve the code. This is less efficient, because the code may reside on a node far away, and therefore, is only used when the first method fails to yield code.

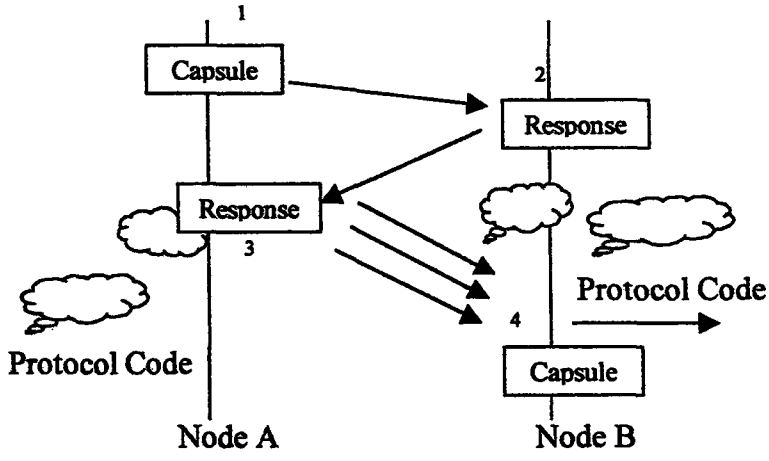

Figure 3.3 Demand Loading Protocol [11]

3.2.4 Routing

Routing in ANTS works much the same way as routing in the Internet today. It uses an algorithm known as "next-hop" routing [27]. Because the Internet is large, it is not feasible to keep track of all paths from every possible source to every possible destination. Not only would this consume too much space, but also it would take too much time to search through the tables to route a packet. As a result, routing tables do not have information about the exact location of all nodes. Instead, they keep track of some nodes, usually those within the same local area network. The routing tables contain a default "next-hop" address for addresses that the router does not know how to get to. When a packet arrives and needs to be routed, the routing table is queried and the result is the address of the next node to which the packet should be sent along the way to its final destination. The list of routers, or nodes, from the source to the destination is the route, or path.

In a TCP/IP network, as well as ANTS, each packet is routed independently [see Section 1.1.1]. This means that different packets from source A may take different paths to destination B depending on network conditions, e.g., congestion or broken links. In addition, routing in TCP/IP networks and ANTS is asymmetric, which means that routes from A to B may not be the same as routes from B to A. This is especially true in the Internet today. Asymmetric routing, however, adds complexity to the network and makes network troubleshooting difficult.

CHAPTER 4

DESIGN AND IMPLEMENTATION

This chapter describes the design of KeyNote trust-management system and provides information about the current implementation of KeyNote for ANTS. Section 4.1 provides an overview of the KeyNote structure, Section 4.2 provides the architecture overview of our implementation, Section 4.3 provides information about principals, Section 4.4 provides information about signatures, Section 4.5 provides information about actions, Section 4.6 provides information about assertions, Section 4.7 explains steps of initializing trust-management environment, Section 4.8 explains steps of creating action attribute set, Section 4.9 provides information about how to process action attribute sets. Finally, in Section 4.10, AssertionPad project written for assertion management purposes are explained.

The keys and signatures shown in figures are not real examples. We use them just to show the syntax of KeyNote and increase the readibility of figures. The Signatures in these figures do not represent the result of any real signature calculation.

4.1 Overview of KeyNote Structure

In KeyNote, the authority to perform trusted actions is associated with one or more "principals". Principals perform two functions:

- They request "actions", any trusted operations that an application places under control and
- They issue "assertions" to delegate the authorization to perform actions to other principals.

Actions are described in terms of a collection of name-value pairs called an "action attribute set". The action attribute set is created by the invoking application. Actions are described in detail in Section 4.5.

Assertions are the basic programming unit for specifying policy and delegating authority. Assertions describe the conditions under which a principal authorizes actions requested by other principals. An assertion identifies the principal that made it, which other principals are being authorized, and the conditions under which the authorization applies. There is two-type assertion:

- Policy Assertion: A special principal, whose identifier is "POLICY", provides the root of trust in KeyNote. Therefore, "POLICY" is considered as authorized to perform any action. Assertions issued by the "POLICY" principal are called "policy assertions" which are used to delegate authority to other untrusted principals. Policy assertions are written to a policy file. There can be more than one policy assertion in this file.
- Credential Assertion: When a public key identifies a principal, this principal can digitally sign assertions and distribute them over untrusted networks for use by other applications. These signed assertions are called "credential assertions", 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.

Assertions are described in detail in Section 4.6.

KeyNote provides advice to applications about the interpretation of policy with regard to specific requested actions. Applications invoke the KeyNote compliance checker by issuing a proposed action attribute set. The KeyNote system determines and returns an appropriate "policy compliance value" from a priority set of possible responses.

The 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 of compliance values at application initialize time.

4.2 Architecture Overview

In our architecture, constructing trust-management environment is implemented in two projects. The first project named "keynote" is implemented for authentication and authorization purposes. The second project named "assertionpad" is implemented for assertion management purposes.

An application writer uses keynote project at application initialize time for,

- Parsing policy assertions from policy file,
- Setting possible responses of policy assertions with their priorities,
- Setting public and private cryptographic keys of application for creating action attribute set and credential assertions by signing them,
- Setting public cryptographic keys which application has right to have for giving rights to other principals while creating credential assertions,
- Setting action attributes of application for creating action attribute set.

An application writer uses keynote project at application runtime for,

- Creating an action attribute set,
- Creating credential assertions,
- Signing action attribute set,
- Authentication of capsule by verifying action attribute set,
- Parsing and verifying credential assertions,
- Authorization of action attribute set.

An application writer uses assertionpad project for,

- Writing assertions for policy file,
- Creating cryptographic keys for assertions,
- Creating ciphers for encrypting and decrypting assertion and key files.

General flow of a capsule between two active applications that run on active nodes is shown in Figure 4.1.

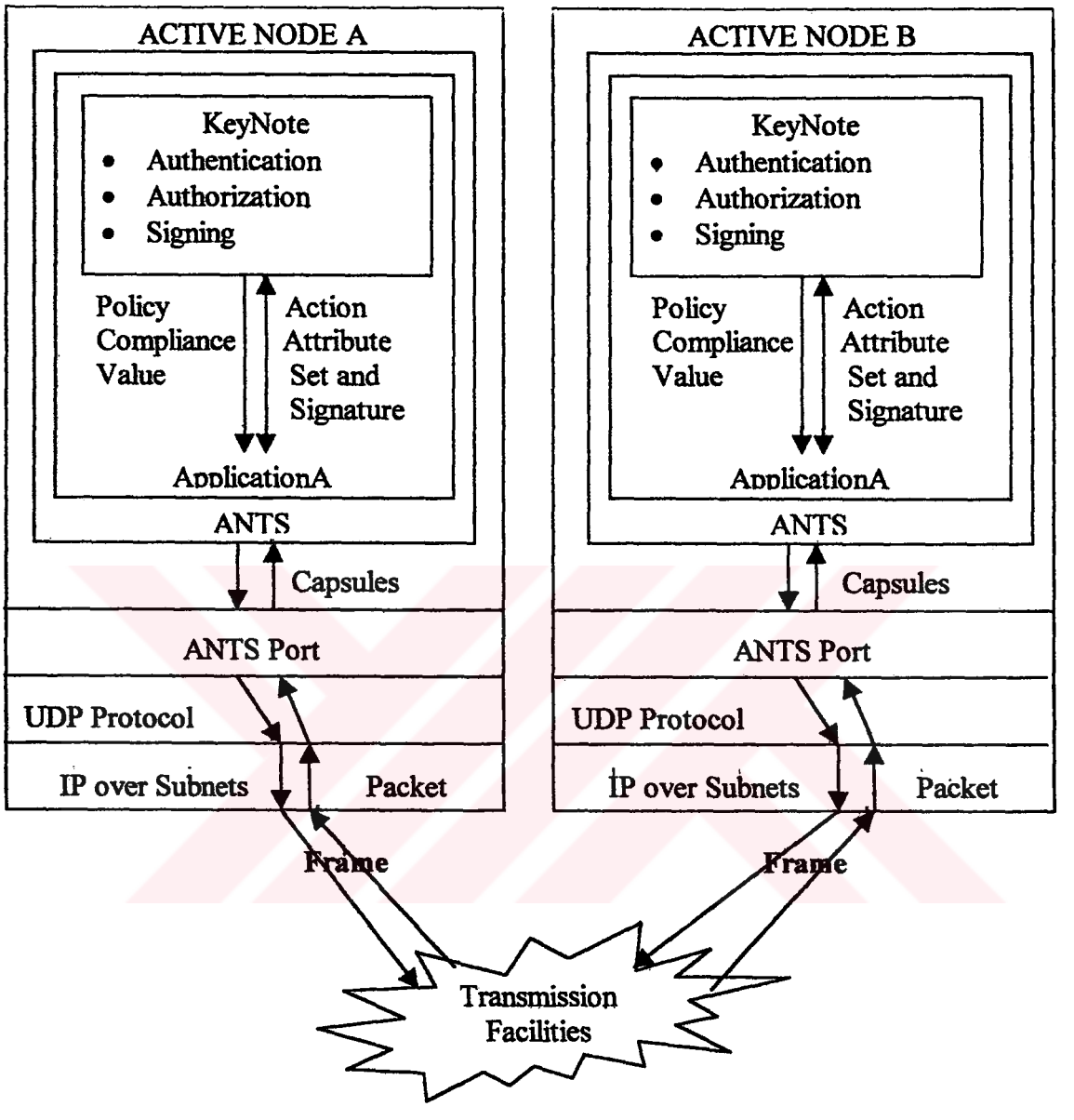

Figure 4.1 KeyNote Architecture

At runtime, first of all, an action attribute set is prepared. The public key of application is added to action attribute set then signed by application private key for authentication purposes. The signature is added to end of action attribute set. Action attribute set is added to capsule. Capsule is send across application link to another application elsewhere in the network. Once in the node, the accept method of the code object associated with the capsule is executed by the node. This method might compare the current node address with the destination address stored within the

capsule's data body. On determining that the capsule is not yet at its destination, the capsule requests that the node send it across a network link towards its destination. This process continues at the nodes along the way until the capsule reaches its destination. At that point, the evaluating capsule checks in the node's state whether a local application is associated with the capsule's code object. If there is such an application, the capsule asks the node to send the capsule across the application link to the application. If this application receives the capsule, it gets and parses action attribute set to get public key, signature and action attributes. It uses public key and signature to authenticate capsule by verifying action attribute set. If authentication result is fail, capsule is ignored; else action attribute set is processed to get policy compliance value. If the policy compliance value is acceptable for application, application process the data comes with capsule else ignore capsule, thus the capsule's journey completes.

4.3 Principals

Principals are represented as ASCII strings called "Principal Identifiers". A Principal Identifier is an arbitrary label whose structure is interpreted by our implementation. Syntax and BNF of principal identifiers are described in Appendix A.1.

4.3.1 Opaque Principals

Principal Identifiers that are used by KeyNote only as labels are said to be "opaque". Opaque identifiers can be only in assertions. They are encoded as strings and every string must correspond to a Cryptographic Principal Identifier in assertions' Local-Constants field except "POLICY" principal identifier [see Section 4.6 for details]. They are used only for increasing the readability of assertions by naming cryptographic principal identifiers.

4.3.2 Cryptographic Principals

Principal identifiers that are used as keys are called cryptographic principal identifiers and they are also lexically encoded as strings. They are used for:

- Identifying the principals authorized by an assertion,
- Identifying the principals that want actions in an action attribute set,

- Signing and verifying credential assertions and action attribute sets for authentication purposes.

In our implementation. a cryptographic principal identifier is implemented by Key class. Every key is converted to a normalized canonical form for the purposes of any internal comparisons between them and all comparisons occur between normalized forms. Keys are encoded as an ASN.1 (Abstract Syntax Notation One) [28] SEQUENCE. For encoding and decoding of keys as ASN.1, ASNContext class is implemented. For using keys in assertions and action attribute sets, an ASN.1 encoded key is then ASCII-encoded as a string of hex digits or base64 characters. For this encoding and decoding, Code class is implemented. Code.bin2Hex and Code.bin2Base64 methods are used for encoding and Code.hex2Bin and Code.base642Bin methods are used for decoding. There are four-type of Key: DSAPublicKey, RSAPublicKey, DSAPrivateKey and RSAPrivateKey. All of them must implement compare and encode method. DSAPrivateKey and RSAPrivateKey must implement sign method too. Key class hierarchy is shown in Figure 4.2.

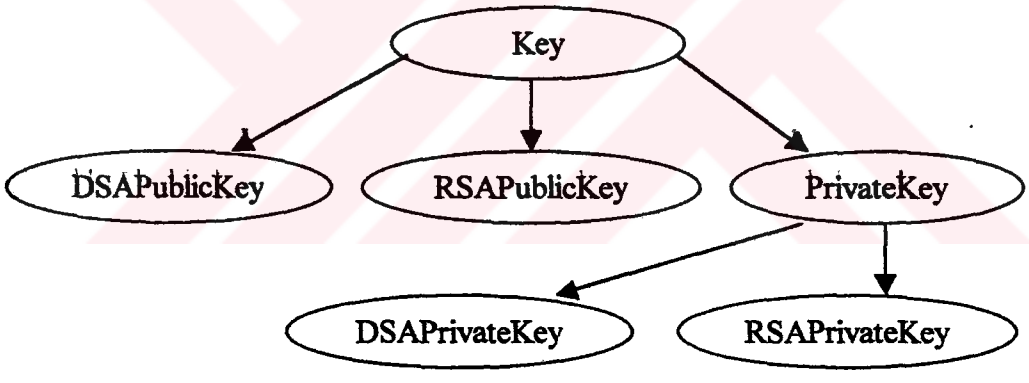

Figure 4.2 Key Class Hierarchy

DSA Public Keys

DSA [29] public keys are implemented by DSAPublicKey class and they are identified by four values: the public value y, the p parameter, the q parameter, and the g parameter. Where the y, p, q, and g are the DSA public key parameters corresponding to the notation of [30]. These four values together make up the DSA key normalized form. All DSA public key comparisons in our implementation occur between normalized forms.

DSA public keys in our implementation are encoded as an ASN.1 SEQUENCE of four ASN.1 INTEGER objects. The four INTEGER objects are the public value y, the p, the q, and the g parameters of the DSA public key, in that order. ASNContext.encodeDSAPublicKey and ASNContext.decodeDSAPublicKey methods are used for encoding and decoding. For use in assertions and action attribute sets, the ASN.1 SEQUENCE is then ASCII-encoded as a string of hex digits or base64 characters. DSA public keys encoded in this way in our implementation is identified by the "dsa-XXX:" algorithm name, where XXX is an ASCII encoding ("hex" or "base64"). An encoded DSA Public Key is shown in Figure 4.3.

```
dsa-hex:3048024100d15d08ce7d2103d93ef21a87
dsa-base64:MEgCQQCzxWCi619s3Bqf8QOZTR
```

Figure 4.3 Encoded DSA Public Key

RSA Public Keys

RSA [31] public keys are implemented by RSAPublicKey and they are identified by two values: the public exponent, and the modulus. These two values together make up the RSA public key normalized form. All RSA public key comparisons in our implementation occur between normalized forms.

RSA public keys in our implementation are encoded as an ASN.1 SEQUENCE of two ASN.1 INTEGER objects. The two INTEGER objects are the public exponent and the modulus of the RSA key, in that order. ASNContext.encodeRSAPublicKey and ASNContext.decodeRSAPublicKey methods are used for encoding and decoding. For use in assertions and action attribute sets, the ASN.1 SEQUENCE is then ASCII-encoded as a string of hex digits or base64 characters. RSA public keys encoded in this way is identified by the "rsa-XXX:" algorithm name, where XXX is an ASCII encoding ("hex" or "base64"). An encoded RSA Public Key is shown in Figure 4.4.

```
rsa-hex:3048024100d15d08ce7d2103d93ef21a87
rsa-base64:MEgCQQCzxWCi619s3Bqf8QOZTR
```

Figure 4.4 Encoded RSA Public Key

DSA Private Keys

DSA [29] private keys are implemented by DSAPrivateKey class and they are identified by six values: the version, the public value y, the p parameter, the q parameter, the g parameter and the private value x. Where the y, p, q, g and x are the DSA public key parameters corresponding to the notation of [30]. These six values together make up the DSA private key normalized form. All DSA private key comparisons in our implementation occur between normalized forms.

DSA private keys in our implementation are encoded as an ASN.1 SEQUENCE of six ASN.1 INTEGER objects. The six INTEGER objects are the public value y, the p, the q, and the g parameters of the DSA private key, in that order. ASNContext.encodeDSAPrivateKey and ASNContext.decodeDSAPrivateKey methods are used for encoding and decoding. For use in assertions and action attribute sets, the ASN.1 SEQUENCE is then ASCII-encoded as a string of hex digits or base64 characters. DSA private keys encoded in this way in our implementation is identified by the "private-dsa-XXX:" algorithm name, where XXX is an ASCII encoding ("hex" or "base64"). An encoded DSA Private Key is shown in Figure 4.5.

```
private-dsa-hex:3048024100d15d08ce7d2103d93
private-dsa-base64:MEgCQQCzxWCi619s3Bqf8
```

Figure 4.5 Encoded DSA Private Key

RSA Private Keys

RSA [31] private keys are implemented by RSAPrivateKey class and they are identified by four values: the version, the public exponent, the modulus and the private exponent. These four values together make up the RSA private key normalized form. All RSA private key comparisons in our implementation occur between normalized forms.

RSA private keys in our implementation are encoded as an ASN.1 SEQUENCE of four ASN.1 INTEGER objects. The four INTEGER objects are the version, the public exponent, the modulus and the private exponent, in that order. ASNContext.encodeRSAPrivateKey and ASNContext.decodeRSAPrivateKey methods are used for encoding and decoding. For use in assertions and action attribute sets, the ASN.1 SEQUENCE is then ASCII-encoded as a string of hex digits

or base64 characters. RSA private keys encoded in this way is identified by the "private-rsa-XXX:" algorithm name, where XXX is an ASCII encoding ("hex" or "base64"). An encoded RSA Private Key is shown in Figure 4.6.

```
private-rsa-hex:3048024100d15d08ce7d2103d93
private-rsa-base64:MEgCQQCzxWCi619s3Bqf8
```

Figure 4.6 Encoded RSA Private Key

4.4 Signatures

In our implementation, signatures are used for authentication purposes. This authentication is done by verifying action attribute sets and credential assertions. A signature is implemented by Signature class. Every signature is converted to a normalized canonical form for authentication and comparison purposes. Signatures are also encoded as an ASN.1 format. For encoding and decoding of signatures as ASN.1, ASNContext class is used also. For using signatures in credential assertions and action attribute sets, an ASN.1 encoded signature is also then ASCII-encoded as a string of hex digits or base64 characters. For this encoding and decoding, Code class is used also. Code.bin2Hex and Code.bin2Base64 methods are used for encoding and Code.hex2Bin and Code.base642Bin methods are used for decoding. While signature computation, SHA1 [32] and MD5 [33] hash algorithms are used. SHA1 algorithm is implemented by SHA class and MD5 algorithm is implemented by MD5 class. There is two-type of Signature: DSASignature and RSASignature. Both of them must implement verify, compare and encode methods. Signature class hierarchy is shown in Figure 4.7.

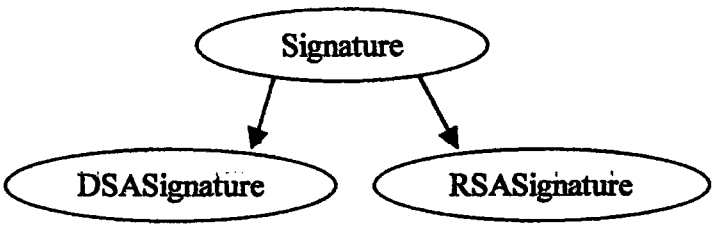

Figure 4.7 Signature Class Hierarchy

4.4.1 DSA Signatures

DSA [29] signatures are implemented by DSASignature class and they are identified by two values: the r value and the s value. These two values together make up the DSA signature normalized form. All DSA signature verifications and comparisons in our implementation occur between normalized forms.

DSA signatures in our implementation are encoded as an ASN.1 SEQUENCE of two ASN.1 INTEGER objects. The two INTEGER objects are the r and s values of a DSA signature [29], in that order. ASNContext.encodeDSASignature and ASNContext.decodeDSASignature methods are used for encoding and decoding. For use in credential assertions and action attribute sets, the ASN.1 SEQUENCE is then ASCII-encoded as a string of hex digits or base64 characters. DSA signatures encoded in this way is identified by the "sig-dsa-XXX-YYY:" algorithm name, where XXX is a hash function name ("sha1", for the SHA1 [32] hash function is currently the only hash function that may be used with DSA) and YYY is an ASCII encoding ("hex" or "base64"). An encoded DSA Signature is shown in Figure 4.8.

```
sig-dsa-sha1-hex:3048024100d15d08ce7d2103d9
sig-dsa-sha1-base64:MEgCQQCzxWCi619s3Bqf
```

Figure 4.8 Encoded DSA Signature

4.4.2 RSA Signatures

RSA [31] signatures are implemented by RSASignature class and they are identified by one value as signature itself. All RSA signature verifications and comparisons in our implementation occur between signatures.

Before signature computation, hash values of assertions or action attribute sets are encoded as an ASN.1 OCTET STRING objects and then PKCS#1 padding is added to this ASN.1 OCTET STRING object. This encoding and decoding are done in RSAPrivateKey.sign and ASNContext.decodeRSASignature methods. After this encoding, signature is computed. For use in credential assertions and action attribute sets, signature is then ASCII-encoded as a string of hex digits or base64 characters. RSA signatures encoded in this way are identified by the "sig-rsa-XXX-YYY:" algorithm name, where XXX is a hash function name, "md5" or "sha1", for the MD5 [33] and SHA1 [32] hash algorithms respectively, may be used with RSA, and YYY

is an ASCII encoding "hex" or "base64". An encoded RSA Signature is shown in Figure 4.9.

```
sig-rsa-sha1-hex:3048024100d15d08ce7d2103d9
sig-rsa-sha1-base64:MEgCQQCzxWCi619s3Bqf
sig-rsa-md5-hex:3048024100d15d08ce7d2103d9
sig-rsa-md5-base64:MEgCQQCzxWCi619s3Bqf
```

Figure 4.9 Encoded RSA Signature

4.5 Actions

Trusted actions to be evaluated by KeyNote are described by a collection of name-value pairs called the "action attribute set". Action attributes are the primary objects on which KeyNote assertions operate. An action attribute set is passed to the KeyNote compliance checker with each request.

Every action attribute set must begin with a set of public keys to define the principals directly authorizing actions and end with computed signature of action attribute set by using application private key for authentication. There can be one or more public keys in this set, so first public key of collection is used to verify signature. Because of this, this collection must start with application's public key.

Between, public key set and signature, an action attribute set consists action attributes. An action attribute is a name and value pair. Arbitrary-length strings represent action attribute names and values. They are the actions that the principals want to perform. Syntax and BNF of action attributes are defined in Appendix A.2. An action attribute set is shown in Figure 4.10.

```
"dsa-hex: 3048024100d, rsa-hex: 15d08ce7d2103d9"
app_domain = "ping application"
some_num = "1"
some_var = "some value"
another_var = "foo"
"sig-dsa-sha1-base64:MEgCQQCzxWCi619s3Bqf"
```

Figure 4.10 Action Attribute Set

4.6 Assertions

Assertions are the basic programming unit for specifying policy and delegating authority. Assertions describe the conditions under which a principal authorizes actions requested by other principals. An assertion identifies the principal that made it, which other principals are being authorized, and the conditions under which the authorization applies.

In our implementation, assertions are implemented by Assertion class and AssertionParser class is implemented for parsing assertions. There is two- type Assertion: PolicyAssertion and CredentialAssertion. Both of them must implement create and getHashValue methods. Policy and Credential assertions are described in more detail in Section 4.6.2 and 4.6.3. Assertion class hierarchy is shown in Figure 4.11. Syntax and BNF of assertions are described in Appendix A.3.

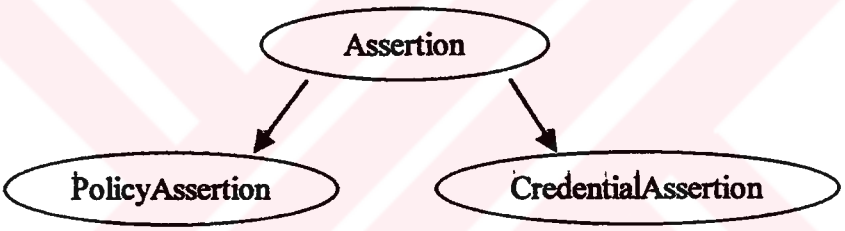

Figure 4.11 Assertion Class Hierarchy

4.6.1 Assertion Fields

Assertions are divided into sections, called "fields", that serve various semantic functions. Each field starts with an identifying label at the beginning of a line, followed by the ':' character and the field's contents. There can be at most one field per line.

One mandatory field is required in all assertions: Authorizer. Six optional fields may also appear: Comment, Conditions, KeyNote-Version, Licensees, Local-Constants and Signature. All field names are case-insensitive. The "KeyNote-Version" field, if present, appears first. The "Signature" field, if present, appears last. Otherwise, fields may appear in any order. Each field may appear at most once in any assertion.

Public keys written in assertion fields are created by using Generator.generateKey method. Signature written in Signature field is created by using Generator.generateSignature method.

The KeyNote-Version Field

The KeyNote-Version field identifies the version of the KeyNote assertion language under which the assertion was written. The KeyNote-Version field, if included, should appear first. Current KeyNote version for our implementation is equal to 2. BNF of this field is described in Appendix A.4. An example of this field is shown in Figure 4.12.

```
KeyNote-Version: 2
```

Figure 4.12 KeyNote-Version Field

The Authorizer Field

The Authorizer identifies the Principal issuing the assertion. If assertion is a policy assertion, its value must be equal to "Policy". If assertion is a credential assertion, its value can be a public key or an opaque principal identifier defined in Local-Constants field as a public key.

BNF of this field is described in Appendix A.6. An example of this field is shown in Figure 4.14.

```
# Authorizer field for policy assertions
Authorizer: "Policy"
# Authorizer field for credential assertions
Authorizer: "dsa-base64: MEgCQQCzxWCi619s3Bqf8"
```

Figure 4.13 Authorizer Field for Policy and Credential Assertions

The Licensees Field

The Licensees field identifies the principals authorized by the assertion. More than one principal can be authorized, and authorization can be distributed across several principals through the use of "and" threshold constructs.

Licensees field can have operators. Because of this, to compute Licensees compliance value of an assertion, this field is converted to postfix format. PostfixConverter class does this conversion. The operator presidencies used while this conversion are listed in Appendix B.

BNF of this field is described in Appendix A.7. An example of this field is shown in Figure 4.15.

```
Licensees: "dsa-base64: MEgCQQCzxWCi619s3Bqf8" ||
            ("rsa-hex:3048024100d15d08ce7d2103d9" &&
             "dsa-hex:b8e57f7ade4a2a31e43017c383ab2a")
```

Figure 4.14 Licensees Field

The Local-Constants Field

This field adds or overrides action attributes in the current assertion only. This mechanism allows the use of short names (opaque principal identifiers [see Section 4.3.1]) for (frequently lengthy) cryptographic principal identifiers, especially to make the Licensees field more readable. If the Local-Constants field defines more than one identifier, it can occupy more than one line and be indented. Attributes defined in the Local-Constants field override any attributes with the same name passed in with the action attribute set. An attribute may be initialized at most once in the Local-Constants field. If an attribute is initialized more than once in an assertion, the entire assertion is considered invalid and is not considered by the KeyNote compliance checker in evaluating action attribute sets.

While creating Assertion class, if assertion has this field, first of all, this field is parsed. LocalConstantParser class is implemented for parsing Local-Constants field. After parsing Local-Constants field, Authorizer and Licensees fields are updated.

BNF of this field is described in Appendix A.5. An example of this field is shown in Figure 4.13.

```
Authorizer: ADMIN
Licensees: TARIK || (OGUZ && KEMAL)
Local-Constants:
    ADMIN = "dsa-hex:63a7d627f13a385b0240581129"
    TARIK  = "dsa-base64: MEgCQQCzxWCi619s3Bqf8"
    OGUZ = "rsa-hex:3048024100d15d08ce7d2103d9"
    KEMAL = "dsa-hex:b8e57f7ade4a2a31e43017c383aba"
```

Figure 4.15 Local-Constants Field

The Conditions Field

This field gives the "conditions" under which the Authorizer trusts the Licensees to perform an action. "Conditions" are predicates that operate on the action attribute set. Conditions field can have operators. Because of this, to compute conditions compliance value of an assertion, this field is converted to postfix format. PostfixConverter class does this conversion. The operator presidencies used while this conversion are listed in Appendix B. Syntax and BNF of this field is described in Appendix A.8. An example of this field is shown in Figure 4.16.

```
Conditions: app_domain == "ping application" &&
            @some_num == 1 && (some_var == "some value"
            || some_var == "some other value") -> "true";
```

Figure 4.16 Conditions Field

The Comment Field

The Comment field allows assertions to be annotated with information describing their purpose. No interpretation of the contents of this field is performed by KeyNote. Note that this is one of two mechanisms for including comments in KeyNote assertions; comments can also be inserted anywhere in an assertion's body by preceding them with the '#' character (except inside string literals). BNF of this field is described in Appendix A.9. An example of this field is shown in Figure 4.17.

```
Comment: This is our first policy assertion
```

Figure 4.17 Comment Field

The Signature Field

The Signature field identifies a signed assertion (credential assertion) and gives the encoded digital signature of the principal identified in the Authorizer field. The algorithm name should be the same as that of the principal appearing in the Authorizer field. It is not necessary that the encodings of the signature and the authorizer key be the same. If the signature field is included, the principal named in the Authorizer field must be a Cryptographic Principal Identifier, the algorithm must be known to the KeyNote implementation, and the signature must be correct for the

assertion body and authorizer key. The signature is always the last field in a KeyNote assertion. Text following this field is not considered part of the assertion. See Section 4.4 for implementation. BNF of this field is described in Appendix A.10. An example of this field is shown in Figure 4.18.

```
Signature: "sig-rsa-sha1-base64:E2OhrczI0LtAYAoJ6fSlqvl \
            QDA4rGiIX73T6p9eExpyHZbfjxPxXEIf6tbBre6x2"
```

Figure 4.18 Signature Field

4.6.2 Policy Assertions

A special principal, whose identifier is "POLICY", provides the root of trust in KeyNote. Therefore, "POLICY" is considered as authorized to perform any action. Assertions issued by the "POLICY" principal are called "policy assertions" which are used to delegate authority to other untrusted principals. Policy assertions are written to a policy file. There can be more than one policy assertion in this file. Policy assertions are implemented by PolicyAssertion class.

For policy assertions, one mandatory field is required: Authorizer. Five optional fields may also appear: Comment, Conditions, KeyNote-Version, Licensees, and Local-Constants. Policy assertions cannot have Signature field.

An example of policy assertion is shown in Figure 4.19.

```
# this is our first policy assertion
Authorizer: "POLICY"
Licensees: KEY1 || KEY2
Local-Constants:
        KEY1 = "rsa-base64:MEgCQQCzxWCi619s3Bqf8Q \
                DS2EVzAgMBAAE="
        KEY2 = "dsa-base64:MIHfAkEAhRzwrvhbRXIJH+n \
                S+dlCGShsYyB+VjSub7Q=="
Comment: A slightly more complicated policy
Conditions: app_domain == "ping application" && @some_num
            == 1 && (some_var == "some value" || some_var ==
            "some other value") -> "true";
```

Figure 4.19 Policy Assertion

4.6.3 Credential Assertions

When a public key identifies a principal, this principal can digitally sign assertions and distribute them over untrusted networks for use by other applications. These signed assertions are called "credential assertions", 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. Credential assertions are implemented by CredentialAssertion class.

For credential assertions, two mandatory fields are required: Authorizer and Signature. Five optional fields may also appear: Comment, Conditions, KeyNote-Version, Licensees, and Local-Constants.

Signatures for credential assertions are computed over the assertion body, starting from the beginning of the first keyword, up to and including the newline character immediately before the "Signature:" keyword, and the signature algorithm name, including the trailing colon character, e.g., "sig-dsa-sha1-base64:" or "sig-rsa-md5-base64:".

An example of credential assertion is shown in Figure 4.20.

```
# this is our first credential assertion
KeyNote-Version: 2
Authorizer: KEY1
Local-Constants:
      KEY1 = "rsa-base64:MEgCQQCzxWCl619s3Bqf8QOZTR \
                  DS2EVzAgMBAAE="
Licensees: "dsa-hex:3081de02402121e160209f7ecef1b \
              0507f5cffe74ed4f1a"
Conditions: app_domain == "ping application" &&
              another_var == "foo" -> "true";
Signature: "sig-rsa-sha1-base64:E2OhrczI0LtAYAoJ6fSlqvl \
              4IS3tuY2w=="
```

Figure 4.20 Credential Assertion

4.7 Initializing of Trust Management Environment

In our implementation, initialization of trust-management environment is done in five steps. All steps are done by using KeynoteManager class methods. These steps are shown in Figure 4.21.

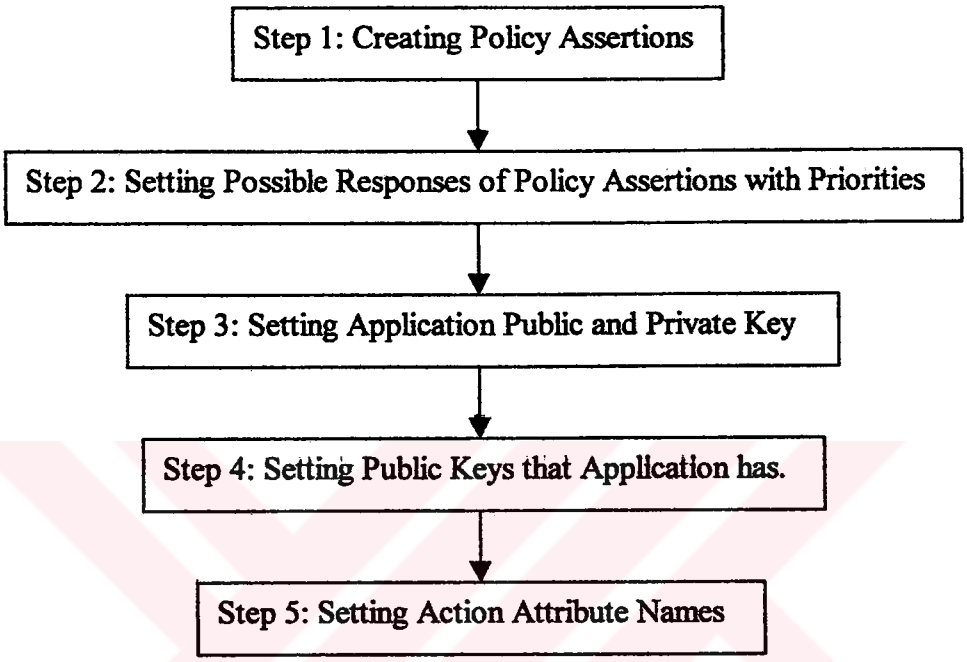

Figure 4.21 Initialization of Trust Management Environment

In first step, Policy assertions are read from policy assertion file and then added to environment by KeynoteManager.processAssertion method. For Reading policy file, parsing policy assertions and creating PolicyAssertion class for each of policy assertion, AssertionParser class is implemented. All three job is done by AssertionParser.readAssertion method and called by KeynoteManager.processAssertion method. Created Policy Assertions are then added to hash table by calling KeynoteManager.addAssertion method. Hash values of assertions are founded by calling Assertion.getHashValue method.

In second step, possible responses of policy assertions are set with priorities. This is done by calling KeynoteManager.addExpectedValue method. For each expected value an ExpectedValue class is created and then added to the end of last added ExpectedValue. While adding ExpectedValue, its existence is controlled, if exist, it is not added.

In third step, private key of application is parsed from a key file for signing action attribute sets and credential assertions. This is done by calling KeynoteManager.setApplicationKey method. After this process, application public and private keys are created.

In fourth step, all public keys which application has rights to have are parsed from a key file. These keys are used for adding public keys to action attribute sets and credential assertions. This is done by calling KeynoteManager.setApplicationPublicKeys method.

In final step, action attribute names are set. These names are used while creating action attribute set. This is done by calling KeynoteManager.setActionAttributeNames method. For preparing an action attribute set, ActionAttributeSet class is implemented and for action attribute names and values, Environment class is implemented. In KeynoteManager.setActionAttributeNames method, first of all, ActionAttributeSet class is created and then for each action attribute names, Environment class is created and added to ActionAttributeSet by calling ActionAttributeSet.add method.

4.8 Creating Action Attribute Set

An action attribute set is created in three steps. These steps can be implemented in a GUI (Graphical User Interface) or application level. Application writer must do this choice. These steps are shown in Figure 4.22.

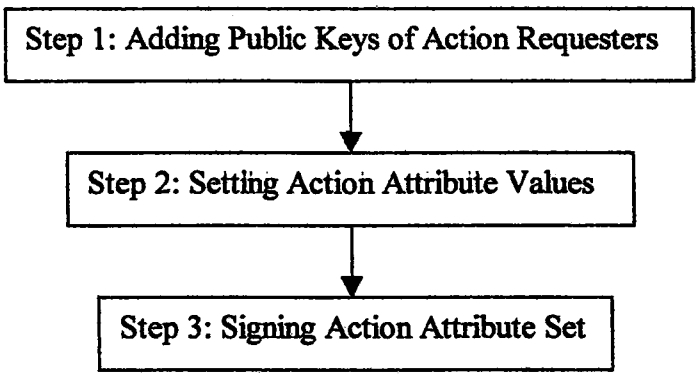

Figure 4.22 Creating Action Attribute Set

In first step, the public keys of action requesters must be added to action attribute set. The first public key must be application public key. Because, the active node

processed action attribute set will use it for authentication. Adding application public key is done internally by calling ActionAttributeSet.prepare method. If any public key is not chosen, only application public key will be added to action attribute set.

In second step, action attribute values of action attribute names must be set. The action attribute names set at initialization phase are used for creating an action attribute set. This setting is done by calling ActionAttributeSet.setValue method.

In third step, the action attribute set is prepared by using public keys and action attributes. The signature algorithm name is added end of this prepared action attribute set (e.g., "sig-dsa-sha1-base64:"), and then signed by application private key. This process is done by calling ActionAttributeSet.prepare method. A prepared action attribute set is shown in Figure 4.2.

```
"dsa-hex: 3048024100d, rsa-hex: 15d08ce7d2103d9"
app_domain = "ping application"
some_num = "1"
some_var = "some value"
another_var = "foo"
"sig-dsa-sha1-base64:MEgCQQCzxWCl619s3Bqf"
```

Figure 4.23 A Prepared Action Attribute Set

4.9 Processing Action Attribute Set

Applications use KeynoteManager.processRequest method to process action attribute set. In this method, action attribute sets are processed in two steps. In first step, action attribute set is authenticated. In second step, action attribute set is authorized. RequestParser class is implemented for parsing action attribute sets.

4.9.1 Authentication – Verifying Action Attribute Set

Authentication of an action attribute set is done by KeynoteManager.authenticateRequest method. This method gets public key of sender application and signature of action attribute set by using RequestParser class. By using public key and signature, verification is done.

4.9.2 Authorization - Policy Compliance Value Calculation

The Policy Compliance Value of an action attribute set is the Principal Compliance Value of the principal named "POLICY". The Compliance Value of a principal <X> is the highest priority of:

- **The Direct Authorization Value of principal <X>:** The Direct Authorization Value of a principal <X> is _MAX_TRUST if <X> is listed in the action attribute set as an authorizer of the action. Otherwise, the Direct Authorization Value of <X> is _MIN_TRUST.
- **The Assertion Compliance Values of all assertions identifying <X> in the Authorizer field:** The Assertion Compliance Value of an assertion is the lowest priority of the assertion's Conditions Compliance Value and its Licensee Compliance Value. For all assertions in assertion hash table, KeynoteManager.evaluateAssertion method is called for finding Conditions and Licensees Compliance Values.

Conditions Compliance Value

The Conditions Compliance Value of an assertion is the highest-order (maximum) value among all successful clauses listed in the conditions section. If no clause's test succeeds or the Conditions field is empty, an assertion's Conditions Compliance Value is considered to be the _MIN_TRUST value. If an assertion's Conditions field is missing entirely, its Conditions Compliance Value is considered to be the _MAX_TRUST value. AssertionEvaluator.evaluateConditions method is used for finding Conditions Compliance value. General rules for computing Conditions Compliance Value is described in Appendix C.1.

Licensee Compliance Value

The Licensee Compliance Value of an assertion is calculated by evaluating the expression in the Licensees field, based on the principal compliance value of the principals named there. If an assertion's Licensees field is empty, its Licensee Compliance Value is considered to be _MIN_TRUST. If an assertion's Licensees field is missing altogether, its Licensee Compliance Value is considered to be _MAX_TRUST. AssertionEvaluator.evaluateLicensees method is used for finding

Licensees Compliance value. General rules for computing Licensees Compliance Value is described in Appendix C.2.

4.10 Assertion Management

AssertionPad project is implemented for Assertion Management purposes. It is a project that has GUI. This project is used for writing assertions (Policy or Credential), creating public and private keys for assertions and creating ciphers to encrypt and decrypt assertions and keys. General user interface is shown in Figure 4.24.

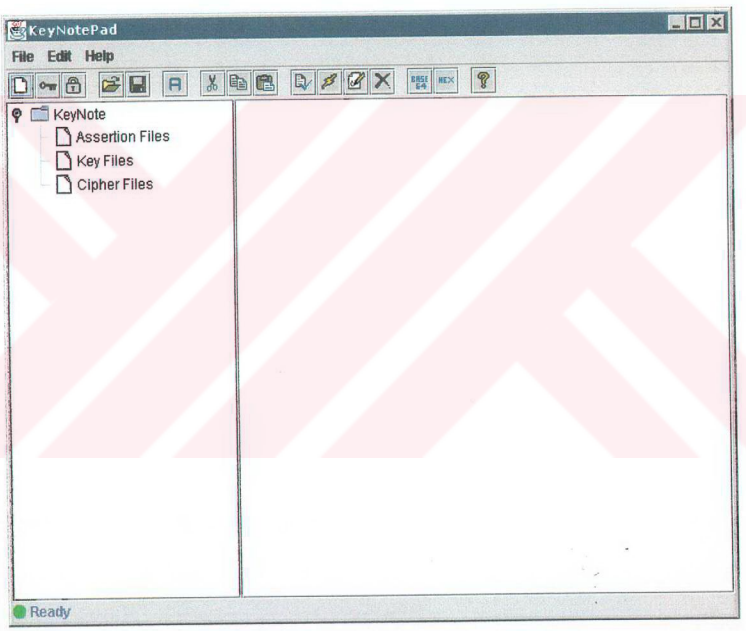

Figure 4.24 General User Interface

4.10.1 Menus

There are three menus: File, Edit, Help.

File Menu

File menu is shown in Figure 4.25. Submenus are described in Table 4.1.

Figure 4.25 File Menu

Table 4.1 Submenus of File Menu

Submenus	**Purpose**
New Assertion	Creating an empty temp assertion file.
New Key	Creating a DSA or RSA public and private key and writing them to a temp key file.
New Cipher	Creating a Triple-DES [34] cipher and writing it to a temp cipher file.
Open	Opening an assertion, key or cipher file.
Close	Closing an assertion, key or cipher file.
Save	Saving an assertion, key or cipher file.
Save as	Saving an assertion, key or cipher file with a different name.
Encrypt	Encrypting an assertion or key file.
Decrypt	Decrypting an encrypted assertion or key file.
Exit	Exiting from program.

Edit Menu

Edit menu is shown in Figure 4.26. Submenus are described in Table 4.2.

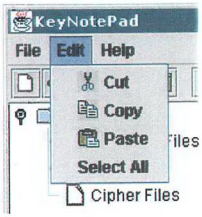

Figure 4.26 Edit Menu

Table 4.2 Submenus of Edit Menu

Submenus	Purpose
Cut	Cut operation for selected text.
Copy	Copy operation for selected text.
Paste	Paste operation.
Select All	Selecting all text.

Help Menu

Help menu is shown in Figure 4.27. Submenus are described in Table 4.3.

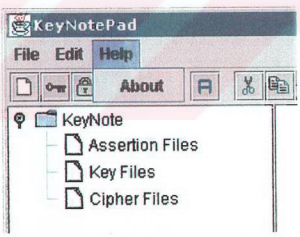

Figure 4.27 Help Menu

Table 4.3 Submenus of Help Menu

Submenus	Purpose
About	Giving information about program.

4.10.2 Toolbar

Toolbar is shown in Figure 4.28. Purposes of these icons are described with respect positions from right to left.

Figure 4.28 Toolbar

Table 4.4 Toolbar Icons

Position	Purpose
1	Creating an empty temp assertion file.
2	Creating a DSA or RSA public and private key and writing them to a temp key file.
3	Creating a Triple-DES [34] cipher and writing it to a temp cipher file.
4	Opening an assertion, key or cipher file.
5	Saving an assertion, key or cipher file.
6	Creating an empty assertion in an assertion file.
7	Cut operation.
8	Copy operation.
9	Paste operation.
10	Compiling an assertion in an assertion file.
11	Building an assertion in an assertion file.
12	Signing an assertion in an assertion file.
13	Deleting an assertion from an assertion file.
14	Changing encoding format of a public and private key in a key file to Base64.
15	Changing encoding format of a public and private key in a key file to hexadecimal.
14	Shows information about program.

4.10.3 Statusbar

Statusbar is used for reporting results of operations. If result is success, a green dot is shown at right side of result text. If result is not success, a red dot is shown.

4.10.4 Popup Menus

All operations can be done via using popup menus. When the right mouse button is clicked over tree nodes, related popup menu is shown. These popup menus are shown in Figure D.1, D.2, D.3, D.4, D.5, D.6, D.7.

4.10.5 Popup Windows

First popup window is shown for choosing algorithm name and encoding format while creating keys. This popup window is shown in Figure 4.36.

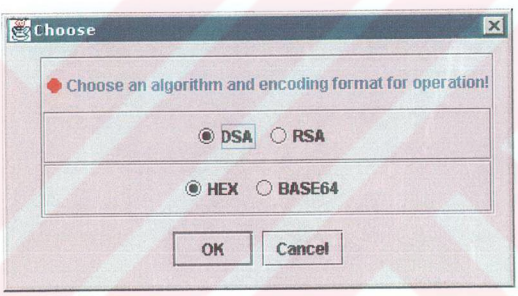

Figure 4.29 Popup Window 1.

Second popup window is shown for choosing cipher while encrypting key or assertion files. This popup window is shown in Figure 4.37.

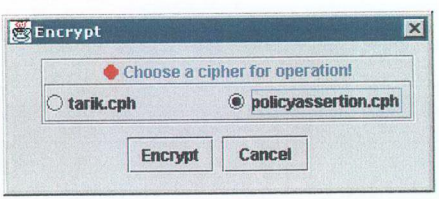

Figure 4.30 Popup Window 2.

Third popup window is shown for choosing cipher while decrypting key or assertion files. This popup window is shown in Figure 4.38.

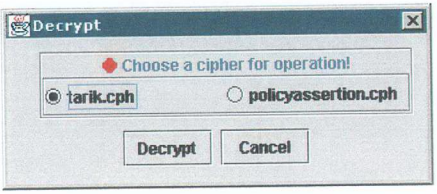

Figure 4.31 Popup Window 3.

Fourth popup window is shown for choosing key while signing an assertion. This popup window is shown in Figure 4.39. If chosen key is RSA key, then a popup window is shown for choosing hash algorithm. This popup window is shown in Figure 4.33. For DSA keys, default and only hash algorithm is SHA1 [32].

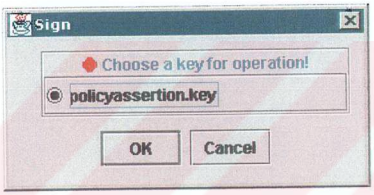

Figure 4.32 Popup Window 4.

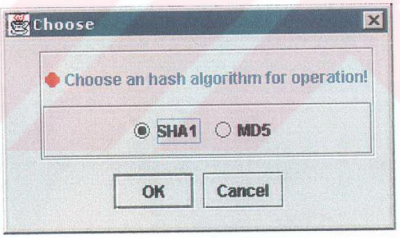

Figure 4.33 Popup Window 5.

CHAPTER 5

CONCLUSION

Active Networks offer the ability to program the network on a per-router, per-user, or even per-packet basis. Unfortunately, this added programmability compromises the security of the system by allowing a wider range of potential attacks. Any feasible Active Network architecture therefore requires strong security guarantees. We have successfully built a KeyNote Trust Management engine in an Active Network, ANTS, to provide security at application level.

5.1 Evaluation

Our goal in this research was to implement KeyNote Trust Management Engine for adding security mechanisms to ANTS. We feel we have met our goal with some success.

- **Authentication:** In KeyNote Trust Management Engine, action attribute sets are used for requesting actions only. We upgrade action attribute sets by signing them for authentication purposes. Our KeyNote Trust Management Engine creates and verifies signatures. By the way, action attribute sets and capsules are authenticated.
- **Authorization:** Once an action attribute set is authenticated, it must be authorized. This authorization process means finding policy compliance value of action attribute set. Our KeyNote Trust Management Engine finds these policy compliance values.

- **Assertion Management:** To facilitate writing assertions, assertionpad project is implemented. This project has a user-friendly GUI (Graphical User Interface). In this project;
    - Assertions can be written,
    - Assertions can be compiled for syntax checking,
    - Assertions can be built by creating an Assertion object,
    - Assertions can be signed for creating credential assertions,
    - Cryptographic keys used by assertions can be created,
    - Ciphers used for encrypting and decrypting assertions and keys can be created,
    - Assertions and keys can be encrypted or decrypted to provide the security.
- **Privilege Levels:** In our implementation, all applications that run on an Active Node must not need to have a KeyNote Trust Management Engine. Thus, Active Network users have access to these services. Application writers must decide on the privilege level of their applications.

5.2 Future Work

Authentication performed in our KeyNote Trust Management Engine has three problems. The first is that our authentication is one-way authentication. The second problem is that there is nothing guarding against replay attacks. Finally, we are using public key operations, which are notoriously slow. To address these problems, we need to implement a protocol based on Diffie-Hellman [35] in which a user and a node authenticate each other and generate a shared secret for future communications.

The KeyNote Trust Management Engine we have built accepts only DSA and RSA keys and Signatures. We need to add new key and signature types (e.g. ElGamal [35], X.509, PGP) to our engine.

The assertionpad project we have built uses Java APIs (Application Programming Interfaces) to create DSA, RSA and Triple-DES keys. These APIs are also used for encryption and decryption of assertions and keys. We need to implement our APIs to

create these keys and encryption and decryption of assertions and keys for backward compatibility.

To further evaluate the effectiveness of our solution, we need to take performance measures on our KeyNote Trust Management Engine in an actual or simulated active network environment. Future investigation would be needed to determine, quantitatively, the impact our KeyNote Trust Management Engine has on system and network performance.

REFERENCES

[1] **Tennenhouse, D.L. et al.**, 1997. A Survey of Active Network Research, *IEEE Communication Magazine*, **35(1)**, pp. 80-86.
http://nms.lcs.mit.edu/activeware/

[2] **Moore, J.T. and Nettles, S.M.**, 2000. Towards Practical Programmable Packets, *Technical Report*, **MS-CIS-00-12**, University of Pennsylvania, Pennsylvania, USA.
http://www.cis.upenn.edu/~switchware/

[3] **Stevens, W.R.**, 1994. TCP/IP Illustrated, Volume 1, Addison-Wesley Publishing Company, Reading, Massachusetts.

[4] **Deering, S.E.**, 1989. Host Extensions for IP Multicasting, Internet RFC 1112.

[5] **Braden, R., Zhang, L., Berson, S. and Jamin, S.** 1997. Resource Reservation Protocol (RSVP) – Version 1 Functional Specification, Internet RFC 2208.

[6] **Atkinson, R.**, 1995. Security Architecture for the Internet Protocol, RFC 1825.

[7] **Heberlein, L.T. and Bishop, M.**, 1996. Attack Class: Address Spoofing, *Proc. 19th National Information Systems Security Conference*, October, pp. 371-377.

[8] **Blaze, M., Feigenbaum, J. and Lacy, J.**, 1996. Decentralized Trust Management, *IEEE Conference on Privacy and Security*, Oakland, USA, pp. 164-173.
http://www.cis.upenn.edu/~angelos/keynote.html

[9] **Blaze, M., Feigenbaum, J. and Strauss, M.**, 1998. Compliance Checking in the PolicyMaker Trust Management System, *Financial Cryptography '98*, Anguilla, BWI , 23-25 February, pp. 254-274.
http://www.cis.upenn.edu/~angelos/keynote.html

[10] **Blaze, M., Feigenbaum J. and Keromytis, A.D.**, 1998. KeyNote: Trust Management for Public-Key Infrastructures, *6th Security Protocols International Workshop*, University of Cambridge, Cambridge, England, 15-17 April, pp. 59-64.
http://www.cis.upenn.edu/~angelos/keynote.html

[11] **Wetherall, D.J., Guttag, J. and Tennenhouse, D.L.**, 1998. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols, *IEEE OPENARCH'98*, San Francisco, USA, April 3-4, pp. 117-129.
http://nms.lcs.mit.edu/activeware/

[12] **Gosling, J., Joy, B. and Steele, G.**, 1996. The Java Language Specification, Addison-Wesley Publishing Company.

[13] **Lindholm, T and Yellin, F.**, 1996. The Java Virtual Machine Specification, Addison-Wesley Publishing Company.

[14] **Rivest, R.**, 1992. The MD5 Message-Digest Algorithm, RFC 1321.

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

[16] **Housley, R., Ford, W., Polk, W. and Solo, D.**, 1999. Internet X.509 Public Key Infrastructure: Certificate and CRL Profile, IETF RFC 2459.

[17] **ISO/IEC 9794-8.** *Information Technology -Open Systems Interconnection- The Directory: Authentication Framework*, 1997. Equivalent to ITU-T Rec. X.509, 1997.

[18] **Gunter, C.A. and Jim, T.** What is QCM?. http://www.cis.upenn.edu/~qcm/

[19] **Gunter, C.A. and Jim, T.**, 2000. Policy-Directed Certificate Retrieval, *SP&E*, **30(15)**, pp. 1609-1640. http://www.cis.upenn.edu/~qcm/

[20] **Gunter, C.A. and Jim, T.**, 2000. Generalized Certificate Revocation, *27th ACM Principles of Programming Languages*, Boston, MA, USA, January 19-21, pp. 316-329. http://www.cis.upenn.edu/~qcm/

[21] **Hicks, M.**, 1998. PLAN System Security, *Technical Report*, **MS-CIS-98-25**, University of Pennsylvania, Pennsylvania, USA. http://www.cis.upenn.edu/~qcm.

[22] **Hicks, M., Kakkar, P., Moore, J.T., Gunter, C.A. and Nettles, S.**, 1998. PLAN: A Packet Language for Active Networks, *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming Languages*, pp. 86-93.

[23] **Hicks, M. and Keromytis, A.D.**, 1999. A Secure Plan, in *Proceedings of The First International Working Conference on Active Networks*, Springer-Verlag, Berlin, Germany, pp. 307-314. http://www.cis.upenn.edu/~switchware/

[24] **Kakkar, P. et al.**, 2000. Certificate Distribution with Local Autonomy, *The Second International Working Conference on Active Networks*, Tokyo, Japan, October 16-18, pp. 277-295. http://www.cis.upenn.edu/~qcm/

[25] **Active Network Backbone (ABONE).** http://www.csl.sri.com/ancors/abone/

[26] **Tennenhouse, D.L. and Wetherall, D.J.**, 1996. Towards an Active Network Architecture, *Computer Communication Review*, **26(2)**, pp. 5-18. http://nms.lcs.mit.edu/activeware/

[27] **Comer, D.E.**, 1995. Internetworking with TCP/IP, Prentice Hall Inc., Englewood Cliffs, NJ, USA.

[28] **Dubuisson, O.**, 2000. ASN.1, Communication Between Heterogeneous Systems, OSA Nokalva, France.

[29] **FIPS PUB 186**, 1994. Digital Signature Standard, *National Institute of Standards and Technology*, USA.

[30] **Schneier, B.**, 1996. Applied Cryptography, John Wiley & Sons, New York, NY.

[31] **Rivest, R.L., Shamir, A. and Adleman, L.M.**, 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Communications of the ACM*, **21(2)**, pp.120-126.

[32] **FIPS PUB 180-1**, 1995. Secure Hash Standard, *National Institute of Standards and Technology*, USA. http://csrc.nist.gov/fips/fip180-1.txt

[33] **Rivest, R.**, 1992. The MD5 Message-Digest Algorithm, RFC 1321.

[34] **The Internet Society**, 1998. The PPP Triple-DES Encryption Protocol (3DESE), RFC 2420.

[35] **Stinson, D.R.**, 1995. Cryptography, Theory and Practice, CRC Press, Inc., Boca Raton, Florida, USA.

APPENDIX A

SYNTAX AND BNF OF KEYNOTE

This Appendix describes the sytax and full BNF of KeyNote. In the following sections, the notation [X]* means zero or more repetitions of character string X. The notation [X]+ means one or more repetitions of X. The notation <X>* means zero or more repetitions of non-terminal <X>. The notation <X>+ means one or more repetitions of X, whereas <X>? means zero or one repetitions of X. Nonterminal grammar symbols are enclosed in angle brackets. Quoted strings in grammar productions represent terminals.

A.1 Principal Identifiers

Opaque principal identifier strings should not contain the ':' character. BNF of principal Identifiers;

<PrincipalIdentifier>:: <OpaqueID> | <KeyID>;

<OpaqueID>:: <StrEx>;

<KeyID>::{see Section 4.3.2};

A.2 Attribute Name and Values

Our implementation guarantees support of attribute names and values up to 2048 characters long. Attribute names begin with an alphabetic or underscore character and they are case-sensitive. Attribute names beginning with the '_' character are reserved for use by the KeyNote runtime environment and cannot be passed from applications as part of queries. In Table A.1, these special attribute names are listed. In addition, attributes with names of the form "_<N>", where <N> is an ASCII-encoded integer, are used by the regular expression matching mechanism. The names of other attributes in the action attribute set are not specified by KeyNote implementation but must be agreed upon by the writers of any policies and credentials that are to inter-operate in a specific KeyNote evaluation.

Attribute values are inherently untyped and are represented as character strings by default. Attribute values may contain any non-NUL ASCII character. Numeric attribute values should first be converted to an ASCII text representation by the invoking application, e.g., the value 1234.5 would be represented by the string "1234.5".

Table A.1 Special Attribute Names.

Name	Purpose
_MIN_TRUST	TRUST Lowest-order (minimum) compliance value in query.
_MAX_TRUST	Highest-order (maximum) compliance value in query.
_VALUES	Compliance values. These values and their priorities are determined by active application.
_ACTION_AUTHORIZERS	Names of principals directly authorizing action in query. Comma separated.

BNF of an action attribute name is;
<AttributeID>:: {Any string starting with a-z, A-Z, or the underscore character,
followed by any number of a-z, A-Z, 0-9, or underscore
characters};

A.3 Assertions

All assertions are encoded in ASCII. A field may be continued over more than one line by indenting subsequent lines with at least one ASCII SPACE or TAB character. Whitespace (a SPACE, TAB, or NEWLINE character) separates tokens but is otherwise ignored outside of quoted strings. Comments with a leading number sign character may begin in any column. Blank lines are not permitted in assertions. Multiple assertions stored in a file (e.g., in application policy file) can be separated from one another unambiguously by the use of blank lines between them. BNF of assertion is;

<Assertion>:: <VersionField>? <AuthField> <LicenseesField>?
<LocalConstantsField>? <ConditionsField>?
<CommentField>? <SignatureField>?;

A.3.1 Comments

The number sign character ('#', ASCII 35 decimal) can be used to introduce comments in assertions. Outside of quoted strings, all characters from the '#' character through the end of the current line are ignored. However, commented text is included in the computation of assertion signatures. Its BNF is;

<Comment>:: '#' {ASCII characters};

A.3.2 Strings

A "string" is a lexical object containing a sequence of characters. Strings may contain any non-NUL characters, including new lines and nonprintable characters. Strings may be given as literals, computed from complex expressions, or dereferenced from attribute names.

String Literals

A string literal directly represents the value of a string. String literals must be quoted by preceding and following them with the double-quote character (ASCII 34 decimal). A printable character may be 'escaped' inside a quoted string literal by preceding it with the backslash character (ASCII 92 decimal) (e.g., "like \"this\"."). This permits the inclusion of the double-quote and backslash characters inside string literals. A similar escape mechanism is also used to represent non-printable characters. '\n' represents the newline character (ASCII character 10 decimal), '\r' represents the carriage-return character (ASCII character 13 decimal), '\t' represents the tab character (ASCII character 9 decimal), and '\f' represents the form-feed character (ASCII character 12 decimal). A backslash character followed by a newline suppresses all subsequent whitespace (including the newline) up to the next non-whitespace character; this allows the continuation of long string constants across lines. Un-escaped newline and return characters are illegal inside string literals. All other escaped characters have the leading backslash removed (e.g., "\a" becomes "a", and "\\" becomes "\"). ;It's BNF;

<StringLiteral>::{Any string between double quoted characters};

String Expressions

In general, anywhere a quoted string literal is allowed, a "string expression" can be used. A string expression constructs a string from string constants, dereferenced attributes, and a string concatenation operator. String expressions may be parenthesized. It's BNF:

```
<StrEx>:: <StrEx> "." <StrEx>        /* String concatenation*/
        | <StringLiteral>
        | "(" <StrEx> ")"
        | <DerefAttribute>
        | "$" <StrEx>;
```

A.3.3 Dereferenced Attributes

Action attributes provide the primary mechanism for applications to pass information to assertions. Attribute names are strings from a limited character set and attribute values are represented internally as strings. An attribute is dereferenced simply by using its name. In general, KeyNote implementation allows the use of an attribute anywhere a string literal is permitted. Attributes are dereferenced as strings by default. When required, dereferenced attributes can be converted to integers or floating point numbers with the type conversion operators '@' and '&'. Thus, an attribute named "foo" having the value "1.2" may be interpreted as the string "1.2" (foo), the integer value 1 (@foo), or the floating point value 1.2 (&foo). Attributes converted to integer and floating point numbers are represented according to the ANSI C "long" and "float" types, respectively. In particular, integers range from 2147483648 to 2147483647, while floats range from 1.17549435E-38F to 3.40282347E+38F. Any uninitialized attribute has the empty-string value when dereferenced as a string and the value zero when dereferenced as an integer or float.

Attribute names may be given literally or calculated from string expressions and may be recursively dereferenced. In the simplest case, an attribute is dereferenced simply by using its name outside of quotes; e.g., the string value of the attribute named "foo" is by reference to "foo" (outside of quotes). The "$<StrEx>" construct dereferences the attribute named in the string expression <StrEx>. For example, if the attribute named "foo" contains the string "bar", the attribute named "bar" contains the string "xyz", and the attribute "xyz" contains the string "qua", the following string comparisons are all true: foo == "bar", $("foo") == "bar", $foo == "xyz", $(foo) == "xyz", $$foo == "qua".

If <StrEx> evaluates to an invalid or uninitialized attribute name, its value is considered to be the empty string (or zero if used as a numeric).

It's BNF:

<DerefAttribute>:: <AttributeID>;

A.4 KeyNote-Version Field

<VersionField>:: "KeyNote-Version:" <VersionString>;

<VersionString>:: <StringLiteral> | <IntegerLiteral>;

A.5 Local-Constants Field

<LocalConstantsField>:: "Local-Constants:" <Assignments>;

<Assignments>:: "" | <AttributeID> "=" <StringLiteral> <Assignments>;

A.6 Authorizer Field

<AuthField>:: "Authorizer:" <AuthID>;

<AuthID>:: <PrincipalIdentifier> | <DerefAttribute> ;

A.7 Licensees Field

<LicenseesField>:: "Licensees:" <LicenseesExpr>;

<LicenseesExpr>:: "" | <PrincExpr>;

<PrincExpr>:: "(" <PrincExpr> ")" | <PrincExpr> "&&" <PrincExpr>
| <PrincExpr> "||" <PrincExpr>
| <K>"-of(" <PrincList> ")" | <PrincipalIdentifier>
| <DerefAttribute>;

<PrincList>:: <PrincipalIdentifier> | <DerefAttribute>
| <PrincList> "," <PrincList>;

<K>:: {Decimal number starting with a digit from 1 to 9};

A.8 Conditions Field

The inability to test for floating point equality, as most floating point implementations (hardware or otherwise) do not guarantee accurate equality testing. Also integer and floating point expressions can only be used within clauses of condition fields, but in no other KeyNote field. The keywords "true" and "false" are not reserved; they can be used as attribute or principal identifier names (although this practice makes assertions difficult to understand and is discouraged).

Regular expression is a standard regular expression, conforming to the POSIX 1003.2 regular expression syntax and semantics.

Any string expression (or attribute) containing the ASCII representation of a numeric value can be converted to an integer or float with the use of the "@" and "&" operators, respectively. Any fractional component of an attribute value dereferenced as an integer is rounded down. If an attribute dereferenced as a number cannot be properly converted (e.g., it contains invalid characters or is empty) its value is considered to be zero.

```
<IntegerLiteral>:: {Decimal number of at least one digit};

<FloatLiteral>:: <IntegerLiteral>"."<IntegerLiteral>;

<ConditionsField>:: "Conditions:" <ConditionsProgram>;

<ConditionsProgram>:: "" | <Clause> ";" <ConditionsProgram>;

<Clause>:: <Test> "(" "{" <ConditionsProgram> "}"
          | <Test> "(" <Value> | <Test>;

<Value>:: <StrEx>;

<Test>:: <RelExpr>;

<RelExpr>:: "(" <RelExpr> ")" | <RelExpr> "&&" <RelExpr>
          | <RelExpr> "||" <RelExpr> | "!" <RelExpr>
          | <IntRelExpr> | <FloatRelExpr> | <StringRelExpr>
          | "true" | "false";

<IntRelExpr>:: <IntEx> "==" <IntEx> | <IntEx> "!=" <IntEx>
            | <IntEx> "<" <IntEx>  | <IntEx> ">" <IntEx>
            | <IntEx> "<=" <IntEx> | <IntEx> ">=" <IntEx>;


<FloatRelExpr>:: <FloatEx> "<" <FloatEx> | <FloatEx> ">" <FloatEx>
              | <FloatEx> "<=" <FloatEx>
              | <FloatEx> ">=" <FloatEx>;

<StringRelExpr>:: <StrEx> "==" <StrEx> | <StrEx> "!=" <StrEx>
               | <StrEx> "<" <StrEx> | <StrEx> ">" <StrEx>
               | <StrEx> "<=" <StrEx> | <StrEx> ">=" <StrEx>
               | <StrEx> "~=" <RegExpr>;
```

<RegExpr>:: {POSIX 1003.2 Regular Expression}

<IntEx>:: <IntEx> "+" <IntEx> | <IntEx> "-" <IntEx>
| <IntEx> "*" <IntEx> | <IntEx> "/" <IntEx>
| <IntEx> "%" <IntEx> | <IntEx> "^" <IntEx>
| "-" <IntEx> | "(" <IntEx> ")" | <IntegerLiteral>
| "@" <StrEx>;

<FloatEx>:: <FloatEx> "+" <FloatEx> | <FloatEx> "-" <FloatEx>
| <FloatEx> "*" <FloatEx> | <FloatEx> "/" <FloatEx>
| <FloatEx> "^" <FloatEx> | "-" <FloatEx>
| "(" <FloatEx> ")" | <FloatLiteral> | "&" <StrEx>;

A.9 Comment Field

<CommentField>:: "Comment:" {Free-form text};

A.10 Signature Field

<SignatureField>:: "Signature:" <Signature>;

<Signature>:: {see Section 4.4}

APPENDIX B

OPERATION PRESEDENCE

This Appendix describes the presedence levels of all operators that are used in KeyNote implementation. In Table B.1, all operators and their presedence levels are listed from lowest to highest. Operators in the same presedence level are evaluated left-to-right.

Table B.1 Presedence Levels of Operators

Operator	Definition	Presedence Level
)	Closed paranthesis	9
}	Closed brace	9
||	Logical OR	8
&&	Logical AND	8
==	Equal to	7
!=	Not Equal to	7
~=	Regular Expression	7
<	Less than	6
<=	Less than or Equal to	6
>	Greater than	6
>=	Greater than or Equal to	6
+	Addition	5
-	Subtraction	5
.	String concatenation	5
*	Multiplication	4
/	Division	4
%	Modulus	4
^	Exponentiation	3
!	Logical NOT	2
-	Unary Minus	2
(	Open Paranthesis	1
{	Open Brace	1

APPENDIX C

COMPLIANCE VALUE CALCULATION

This appendix describes the general rules for computing compliance values for Conditions and Licensees fields of assertions.

C.1 Computing Conditions Compliance Value

The set of successful test clause values is calculated as follows: each clause in the conditions section has two logical parts: a "test" and an optional "value", which, if present, is separated from the test with the "→" token. The test subclause is a predicate that either succeeds (evaluates to logical "true") or fails (evaluates to logical "false"). The value subclause is a string expression that evaluates to one value from the ordered set of compliance values given with the query. If the value subclause is missing, it is considered to be _MAX_TRUST. That is, the clause foo == "bar"; is equivalent to foo=="bar" → _MAX_TRUST. If the value component of a clause is present, in the simplest case it contains a string expression representing a possible compliance value. For example, consider an assertion with the following Conditions field:

```
Conditions: @user_id == 0 → "full_access";          # clause (1)
            @user_id < 1000 → "user_access";        # clause (2)
            @user_id < 10000 → "guest_access";      # clause (3)
            user_name == "root" → "full_access";    # clause (4)
```

Here, if the value of the "user_id" attribute is "1073" and the "user_name" attribute is "root", the possible compliance value set would contain the values "guest_access" (by clause (3)) and "full_access" (by clause (4)). If the priority of compliance values given in the query (in ascending order) is {"no_access", "guest_access", "user_access", "full_access"}, the conditions compliance value of the assertion would be "full_access" (because "full_access" has a higher-priority value than "guest_access"). If the "user_id" attribute had the value "19283" and the "user_name" attribute had the value "nobody", no clause would succeed and the Conditions Compliance Value would be "no_access", which is the lowest-priority value (_MIN_TRUST). If a clause lists an explicit value, its value string must be named in the query ordered compliance value set. Values not named in the query compliance value set are considered equivalent to _MIN_TRUST. The value

component of a clause can also contain recursively-nested clauses. Recursively-nested clauses are evaluated only if their parent test is true. That is,

a=="b" → {b=="c" → "value1";

d=="e" → "value2";

true → "value3"; } ;

is equivalent to

(a=="b") && (b=="c") → "value1";

(a=="b") && (d=="e") → "value2";

(a=="b") → "value3";

String comparisons are case-sensitive. A regular expression comparison ("~=") is considered true if the left-hand-side string expression matches the right-hand-side regular expression. If the POSIX regular expression group matching scheme is used, the number of groups matched is placed in the temporary meta-attribute "_0" (dereferenced as _0), and each match is placed in sequence in the temporary attributes (_1, to _N). These match-attributes' values are valid only within subsequent references made within the same clause. Regular expression evaluation is case-sensitive. A runtime error occurring in the evaluation of a test, such as division by zero or an invalid regular expression, causes the test to be considered false. For example:

foo == "bar" → {@a == 1/0 → "oneval"; # subclause 1

@a == 2 → "anotherval"; # subclause 2};

Here, subclause 1 triggers a runtime error. Subclause 1 is therefore false (and has the value _MIN_TRUST). Subclause 2, however, would be evaluated normally. An invalid <RegExpr> is considered a runtime error and causes the test in which it occurs to be considered false.

C.2 Computing Licensees Compliance Value

For each principal named in the Licensees field, its principal compliance value is substituted for its name. If no Principal Compliance Value can be found for some named principal, its name is substituted with the _MIN_TRUST value. The Licensees expression (as defined in Appendix A) is evaluated as follows:

A "(...)" expression has the value of the enclosed subexpression.

A "&&" expression has the lower-order (minimum) of its two subexpression values.

A "||" expression has the higher-order (maximum) of its two subexpression values.

A "<K>-of(<List>)" expression has the K-th highest order compliance value listed in <list>. Values that appear multiple times are counted with multiplicity. For example, if K = 3 and the orders of the listed compliance values are (0, 1, 2, 2, 3), the value of the expression is the compliance value of order 2.

For example, consider the following Licensees field:

Licensees: ("alice" && "bob") || "eve"

If the principal compliance value is "yes" for principal "alice", "no" for principal "bob", and "no" for principal "eve", and "yes" is higher order than "no" in the query's compliance value set, then the resulting Licensee Compliance Value is "no". Observe that if there are exactly two possible compliance values (e.g., "false" and "true"), the rules of Licensee Compliance Value resolution reduce exactly to standard boolean logic.

APPENDIX D

POPUP MENUS

In this this appendix, figures of popup menus which is implemented in assertionpad project are shown.

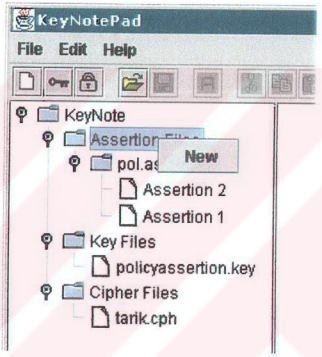

Figure D.1 Popup Menu 1.

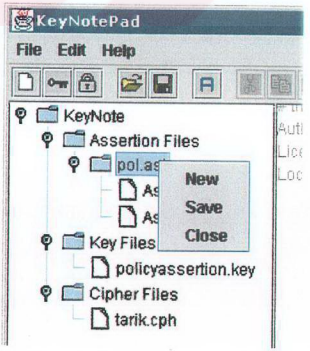

Figure D.2 Popup Menu 2.

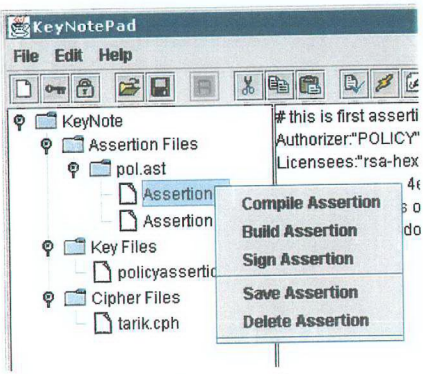

Figure D.3 Popup Menu 3.

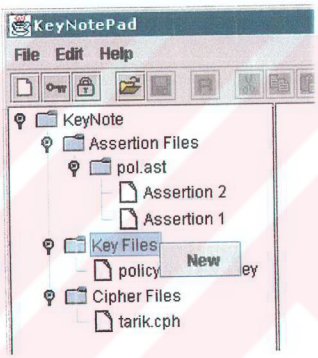

Figure D.4 Popup Menu 4.

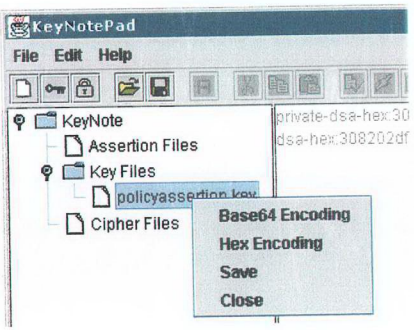

Figure D.5 Popup Menu 5.

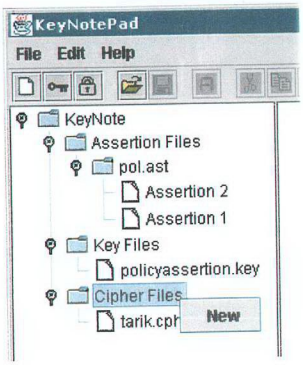

Figure D.6 Popup Menu 6.

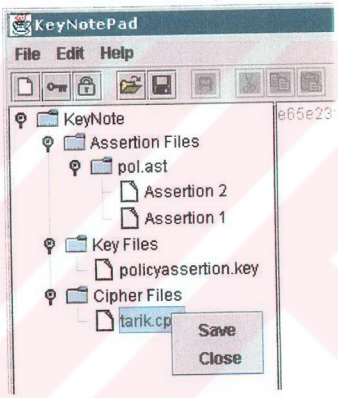

Figure D.7 Popup Menu 7.

BIOGRAPHY

He was born at Çankırı on 1973. He graduated from Naval Lycee on 1991, from Naval Academy, Control and Computer Engineering program on 1995, from Kocaeli University, Master of Business Administration and Organization program on 1998. He has began master education in İstanbul Technical University, Institute of Science and Technology, Control and Computer Engineering program on 1998. He presently works as an officer at Naval Academy.