

Network Working Group  
Request for Comments: 2704  
Category: Informational

M. Blaze  
J. Feigenbaum  
J. Ioannidis  
AT&T Labs - Research  
A. Keromytis  
U. of Pennsylvania  
September 1999

## The KeyNote Trust-Management System Version 2

### Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

### Copyright Notice

Copyright (C) The Internet Society (1999). All Rights Reserved.

### Abstract

This memo describes version 2 of the KeyNote trust-management system. It specifies the syntax and semantics of KeyNote 'assertions', describes 'action attribute' processing, and outlines the application architecture into which a KeyNote implementation can be fit. The KeyNote architecture and language are useful as building blocks for the trust management aspects of a variety of Internet protocols and services.

### 1. Introduction

Trust management, introduced in the PolicyMaker system [BFL96], is a unified approach to specifying and interpreting security policies, credentials, and relationships; it allows direct authorization of security-critical actions. A trust-management system provides standard, general-purpose mechanisms for specifying application security policies and credentials. Trust-management credentials describe a specific delegation of trust and subsume the role of public key certificates; unlike traditional certificates, which bind keys to names, credentials can bind keys directly to the authorization to perform specific tasks.

A trust-management system has five basic components:

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

The trust-management approach has a number of advantages over other mechanisms for specifying and controlling authorization, especially when security policy is distributed over a network or is otherwise decentralized.

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

Trust-management policies are easy to distribute across networks, helping to avoid the need for application-specific distributed policy configuration mechanisms, access control lists, and certificate parsers and interpreters.

For a general discussion of the use of trust management in distributed system security, see [Bla99].

KeyNote is a simple and flexible trust-management system designed to work well for a variety of large- and small-scale Internet-based applications. It provides a single, unified language for both local policies and credentials. KeyNote policies and credentials, called 'assertions', contain predicates that describe the trusted actions permitted by the holders of specific public keys. KeyNote assertions are essentially small, highly-structured programs. A signed

assertion, which can be sent over an untrusted network, is also called a 'credential assertion'. Credential assertions, which also serve the role of certificates, have the same syntax as policy assertions but are also signed by the principal delegating the trust.

In KeyNote:

- \* Actions are specified as a collection of name-value pairs.
- \* Principal names can be any convenient string and can directly represent cryptographic public keys.
- \* The same language is used for both policies and credentials.
- \* The policy and credential language is concise, highly expressive, human readable and writable, and compatible with a variety of storage and transmission media, including electronic mail.
- \* The compliance checker returns an application-configured 'policy compliance value' that describes how a request should be handled by the application. Policy compliance values are always positively derived from policy and credentials, facilitating analysis of KeyNote-based systems.
- \* Compliance checking is efficient enough for high-performance and real-time applications.

This document describes the KeyNote policy and credential assertion language, the structure of KeyNote action descriptions, and the KeyNote model of computation.

We assume that applications communicate with a locally trusted KeyNote compliance checker via a 'function call' style interface, sending a collection of KeyNote policy and credential assertions plus an action description as input and accepting the resulting policy compliance value as output. However, the requirements of different applications, hosts, and environments may give rise to a variety of different interfaces to KeyNote compliance checkers; this document does not aim to specify a complete compliance checker API.

## 2. KeyNote Concepts

In KeyNote, the authority to perform trusted actions is associated with one or more 'principals'. A principal may be a physical entity, a process in an operating system, a public key, or any other convenient abstraction. KeyNote principals are identified by a string called a 'Principal Identifier'. In some cases, a Principal Identifier will contain a cryptographic key interpreted by the

KeyNote system (e.g., for credential signature verification). In other cases, Principal Identifiers may have a structure that is opaque to KeyNote.

Principals perform two functions of concern to KeyNote: They request 'actions' and they issue 'assertions'. Actions are any trusted operations that an application places under KeyNote control. Assertions delegate the authorization to perform actions to other principals.

Actions are described to the KeyNote compliance checker in terms of a collection of name-value pairs called an 'action attribute set'. The action attribute set is created by the invoking application. Its structure and format are described in detail in Section 3 of this document.

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 'query' containing a proposed action attribute set and identifying the principal(s) requesting it. The KeyNote system determines and returns an appropriate 'policy compliance value' from an ordered 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 (from 'weakest' to 'strongest') of compliance values at query time.

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. The syntax of assertions is given in Section 4.

A special principal, whose identifier is "POLICY", provides the root of trust in KeyNote. "POLICY" is therefore considered to be authorized to perform any action.

Assertions issued by the "POLICY" principal are called 'policy assertions' and are used to delegate authority to otherwise untrusted principals. The KeyNote security policy of an application consists of a collection of policy assertions.

When a principal is identified by a public key, it can digitally sign assertions and distribute them over untrusted networks for use by other KeyNote compliance checkers. These signed assertions are also called 'credentials', and serve a role similar to that of traditional public key certificates. Policies and credentials share the same syntax and are evaluated according to the same semantics. A principal can therefore convert its policy assertions into credentials simply by digitally signing them.

KeyNote is designed to encourage the creation of human-readable policies and credentials that are amenable to transmission and storage over a variety of media. Its assertion syntax is inspired by the format of RFC822-style message headers [Cro82]. A KeyNote assertion contains a sequence of sections, called 'fields', each of which specifies one aspect of the assertion's semantics. Fields start with an identifier at the beginning of a line and continue until the next field is encountered. For example:

```
KeyNote-Version: 2
Comment: A simple, if contrived, email certificate for user mab
Local-Constants: ATT_CA_key = "RSA:acdfaldf1011bbac"
                  mab_key = "DSA:deadbeefcafe001a"
Authorizer: ATT_CA_key
Licensees: mab_key
Conditions: ((app_domain == "email") # valid for email only
             && (address == "mab@research.att.com"));
Signature: "RSA-SHA1:f00f2244"
```

The meanings of the various sections are described in Sections 4 and 5 of this document.

KeyNote semantics resolve the relationship between an application's policy and actions requested by other principals, as supported by credentials. The KeyNote compliance checker processes the assertions against the action attribute set to determine the policy compliance value of a requested action. These semantics are defined in Section 5.

An important principle in KeyNote's design is 'assertion monotonicity'; the policy compliance value of an action is always positively derived from assertions made by trusted principals. Removing an assertion never results in increasing the compliance value returned by KeyNote for a given query. The monotonicity

property can simplify the design and analysis of complex network-based security protocols; network failures that prevent the transmission of credentials can never result in spurious authorization of dangerous actions. A detailed discussion of monotonicity and safety in trust management can be found in [BFL96] and [BFS98].

### 3. Action Attributes

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 mechanism by which applications communicate requests to KeyNote and are the primary objects on which KeyNote assertions operate. An action attribute set is passed to the KeyNote compliance checker with each query.

Each action attribute consists of a name and a value. The semantics of the names and values are not interpreted by KeyNote itself; they vary from application to application and must be agreed upon by the writers of applications and the writers of the policies and credentials that will be used by them.

Action attribute names and values are represented by arbitrary-length strings. KeyNote guarantees support of attribute names and values up to 2048 characters long. The handling of longer attribute names or values is not specified and is KeyNote-implementation-dependent. Applications and assertions should therefore avoid depending on the use of attributes with names or values longer than 2048 characters. The length of an attribute value is represented by an implementation-specific mechanism (e.g., NUL-terminated strings, an explicit length field, etc.).

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".

Attribute names are of the form:

```
<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} ;
```

That is, an <AttributeID> begins with an alphabetic or underscore character and can be followed by any number of alphanumerics and underscores. Attribute names are case-sensitive.

The exact mechanism for passing the action attribute set to the compliance checker is determined by the KeyNote implementation. Depending on specific requirements, an implementation may provide a mechanism for including the entire attribute set as an explicit parameter of the query, or it may provide some form of callback mechanism invoked as each attribute is dereferenced, e.g., for access to kernel variables.

If an action attribute is not defined its value is considered to be the empty string.

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. The following special attribute names are used:

Name	Purpose
-----	-----
_MIN_TRUST	Lowest-order (minimum) compliance value in query; see Section 5.1.
_MAX_TRUST	Highest-order (maximum) compliance value in query; see Section 5.1.
_VALUES	Linearly ordered set of compliance values in query; see Section 5.1. Comma separated.
_ACTION_AUTHORIZERS	Names of principals directly authorizing action in query. Comma separated.

In addition, attributes with names of the form "\_<N>", where <N> is an ASCII-encoded integer, are used by the regular expression matching mechanism described in Section 5.

The assignment and semantics of any other attribute names beginning with "\_" is unspecified and implementation-dependent.

The names of other attributes in the action attribute set are not specified by KeyNote but must be agreed upon by the writers of any policies and credentials that are to inter-operate in a specific KeyNote query evaluation.

By convention, the name of the application domain over which action attributes should be interpreted is given in the attribute named "app\_domain". The IANA (or some other suitable authority) will provide a registry of reserved app\_domain names. The registry will list the names and meanings of each application's attributes.

The app\_domain convention helps to ensure that credentials are interpreted as they were intended. An attribute with any given name may be used in many different application domains but might have different meanings in each of them. However, the use of a global registry is not always required for small-scale, closed applications; the only requirement is that the policies and credentials made available to the KeyNote compliance checker interpret attributes according to the same semantics assumed by the application that created them.

For example, an email application might reserve the app\_domain "RFC822-EMAIL" and might use the attributes named "address" (the email address of a message's sender), "name" (the human name of the message sender), and any "organization" headers present (the organization name). The values of these attributes would be derived in the obvious way from the email message headers. The public key of the message's signer would be given in the "\_ACTION\_AUTHORIZERS" attribute.

Note that "RFC822-EMAIL" is a hypothetical example; such a name may or may not appear in the actual registry with these or different attributes. (Indeed, we recognize that the reality of email security is considerably more complex than this example might suggest.)

#### 4. KeyNote Assertion Syntax

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.

##### 4.1 Basic Structure

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

All KeyNote assertions are encoded in ASCII.



KeyNote 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.

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 octothorp character (see Section 4.2) may begin in any column.

One mandatory field is required in all assertions:

Authorizer

Six optional fields may also appear:

Comment  
Conditions  
KeyNote-Version  
Licensees  
Local-Constants  
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.

Blank lines are not permitted in assertions. Multiple assertions stored in a file (e.g., in application policy configurations), therefore, can be separated from one another unambiguously by the use of blank lines between them.

## 4.2 Comments

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

The octothorp character ("#", ASCII 35 decimal) can be used to introduce comments. Outside of quoted strings (see Section 4.3), 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 (see Section 4.6.7).

### 4.3 Strings

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

#### 4.3.1 String Literals

`<StringLiteral>:: "\"" {see description below} "\"" ;`

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.

The constructs "\0o", "\0oo", and "\ooo" (where o represents any octal digit) may be used to represent any non-NUL ASCII characters with their corresponding octal values (thus, "\012" is the same as "\n", "\101" is "A", and "\377" is the ASCII character 255 decimal). However, the NUL character cannot be encoded in this manner; "\0", "\00", and "\000" are converted to the strings "0", "00", and "000" respectively. Similarly, all other escaped characters have the leading backslash removed (e.g., "\a" becomes "a", and "\\" becomes "\"). The following four strings are equivalent:

```
"this string contains a newline\n followed by one space."
"this string contains a newline\n \
followed by one space."
```

```
"this str\
  ing contains a \
  newline\n followed by one space."
```

```
"this string contains a newline\012\040followed by one space."
```

#### 4.3.2 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 (described in Section 4.4), and a string concatenation operator. String expressions may be parenthesized.

```
<StrEx>:: <StrEx> "." <StrEx>      /* String concatenation */
          | <StringLiteral>          /* Quoted string */
          | "(" <StrEx> ")"
          | <DerefAttribute>         /* See Section 4.4 */
          | "$" <StrEx> ;             /* See Section 4.4 */
```

The "\$" operator has higher precedence than the "." operator.

#### 4.4 Dereferenced Attributes

Action attributes provide the primary mechanism for applications to pass information to assertions. Attribute names are strings from a limited character set (<AttributeID> as defined in Section 3), and attribute values are represented internally as strings. An attribute is dereferenced simply by using its name. In general, KeyNote 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, whilst 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).

The <DerefAttribute> token is defined as:

```
<DerefAttribute>:: <AttributeID> ;
```

## 4.5 Principal Identifiers

Principals are represented as ASCII strings called 'Principal Identifiers'. Principal Identifiers may be arbitrary labels whose structure is not interpreted by the KeyNote system or they may encode cryptographic keys that are used by KeyNote for credential signature verification.

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

### 4.5.1 Opaque Principal Identifiers

Principal Identifiers that are used by KeyNote only as labels are said to be 'opaque'. Opaque identifiers are encoded in assertions as strings (see Section 4.3):

```
<OpaqueID>:: <StrEx> ;
```

Opaque identifier strings should not contain the ":" character.

#### 4.5.2 Cryptographic Principal Identifiers

Principal Identifiers that are used by KeyNote as keys, e.g., to verify credential signatures, are said to be 'cryptographic'. Cryptographic identifiers are also lexically encoded as strings:

```
<KeyID>:: <StrEx> ;
```

Unlike Opaque Identifiers, however, Cryptographic Identifier strings have a special form. To be interpreted by KeyNote (for signature verification), an identifier string should be of the form:

```
<IDString>:: <ALGORITHM>":"<ENCODEDBITS> ;
```

"ALGORITHM" is an ASCII substring that describes the algorithms to be used in interpreting the key's bits. The ALGORITHM identifies the major cryptographic algorithm (e.g., RSA [RSA78], DSA [DSA94], etc.), structured format (e.g., PKCS1 [PKCS1]), and key bit encoding (e.g., HEX or BASE64). By convention, the ALGORITHM substring starts with an alphabetic character and can contain letters, digits, underscores, or dashes (i.e., it should match the regular expression "[a-zA-Z][a-zA-Z0-9\_-]\*"). The IANA (or some other appropriate authority) will provide a registry of reserved algorithm identifiers.

"ENCODEDBITS" is a substring of characters representing the key's bits, the encoding and format of which depends on the ALGORITHM. By convention, hexadecimal encoded keys use lower-case ASCII characters.

Cryptographic Principal Identifiers are converted to a normalized canonical form for the purposes of any internal comparisons between them; see Section 5.2.

Note that the keys used in examples throughout this document are fictitious and generally much shorter than would be required for security in practice.

### 4.6 KeyNote Fields

#### 4.6.1 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 is of the form

```
<VersionField>:: "KeyNote-Version:" <VersionString> ;
<VersionString>:: <StringLiteral>
                  | <IntegerLiteral> ;
```

where <VersionString> is an ASCII-encoded string. Assertions in production versions of KeyNote use decimal digits in the version representing the version number of the KeyNote language under which they are to be interpreted. Assertions written to conform with this document should be identified with the version string "2" (or the integer 2). The KeyNote-Version field, if included, should appear first.

#### 4.6.2 The Local-Constants Field

This field adds or overrides action attributes in the current assertion only. This mechanism allows the use of short names for (frequently lengthy) cryptographic principal identifiers, especially to make the Licensees field more readable. The Local-Constants field is of the form:

```
<LocalConstantsField>:: "Local-Constants:" <Assignments> ;
<Assignments>:: /* can be empty */
                | <AttributeID> "=" <StringLiteral> <Assignments> ;
```

<AttributeID> is an attribute name from the action attribute namespace as defined in Section 3. The name is available for use as an attribute in any subsequent field. If the Local-Constants field defines more than one identifier, it can occupy more than one line and be indented. <StringLiteral> is a string literal as described in Section 4.3. 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 queries.

#### 4.6.3 The Authorizer Field

The Authorizer identifies the Principal issuing the assertion. This field is of the form

```
<AuthField>:: "Authorizer:" <AuthID> ;
<AuthID>:: <PrincipalIdentifier>
          | <DerefAttribute> ;
```

The Principal Identifier may be given directly or by reference to the attribute namespace (as defined in Section 4.4).

#### 4.6.4 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' and threshold constructs. This field is of the form

```

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

<LicenseesExpr>::      /* can be empty */
                    | <PrincExpr> ;

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

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

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

```

The "&&" operator has higher precedence than the "||" operator. <K> is an ASCII-encoded positive decimal integer. If a <PrincList> contains fewer than <K> principals, the entire assertion is omitted from processing.

#### 4.6.5 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. The Conditions field is of the form:

```

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

<ConditionsProgram>:: /* Can be empty */
                    | <Clause> ";" <ConditionsProgram> ;

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

<Value>:: <StrEx> ;

```

```
<Test>:: <RelExpr> ;
```

```
<RelExpr>:: "(" <RelExpr> ")"          /* Parentheses */
| <RelExpr> "&&" <RelExpr> /* Logical AND */
| <RelExpr> "||" <RelExpr> /* Logical OR */
| "!" <RelExpr>          /* Logical NOT */
| <IntRelExpr>
| <FloatRelExpr>
| <StringRelExpr>
| "true"                /* case insensitive */
| "false" ;             /* case insensitive */
```

```
<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> /* String equality */
| <StrEx> "!=" <StrEx> /* String inequality */
| <StrEx> "<" <StrEx> /* Alphanum. comparisons */
| <StrEx> ">" <StrEx>
| <StrEx> "<=" <StrEx>
| <StrEx> ">=" <StrEx>
| <StrEx> "~=" <RegExpr> ; /* Reg. expr. matching */
```

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

```
<FloatEx>:: <FloatEx> "+" <FloatEx> /* Floating point */
| <FloatEx> "-" <FloatEx>
| <FloatEx> "*" <FloatEx>
| <FloatEx> "/" <FloatEx>
| <FloatEx> "^" <FloatEx> /* Exponentiation */
```



```

| "-" <FloatEx>
| "(" <FloatEx> ")"
| <FloatLiteral>
| "&" <StrEx> ;

```

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

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

<StringLiteral> is a quoted string as defined in Section 4.3

<AttributeID> is defined in Section 3.

The operation precedence classes are (from highest to lowest):

```

{ ( , ) }
{ unary -, @, &, $ }
{ ^ }
{ *, /, % }
{ +, -, . }

```

Operators in the same precedence class are evaluated left-to-right.

Note the inability to test for floating point equality, as most floating point implementations (hardware or otherwise) do not guarantee accurate equality testing.

Also note that 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).

<RegExpr> 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.

#### 4.6.6 The Comment Field

The Comment field allows assertions to be annotated with information describing their purpose. It is of the form

```
<CommentField>:: "Comment:" <text> ;
```

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).

#### 4.6.7 The Signature Field

The Signature field identifies a signed assertion and gives the encoded digital signature of the principal identified in the Authorizer field. The Signature field is of the form:

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

```
<Signature>:: <StrEx> ;
```

The <Signature> string should be of the form:

```
<IDString>:: <ALGORITHM>":"<ENCODEDBITS> ;
```

The formats of the "ALGORITHM" and "ENCODEDBITS" substrings are as described for Cryptographic Principal Identifiers in Section 4.4.2. The algorithm name should be the same as that of the principal appearing in the Authorizer field. The IANA (or some other suitable authority) will provide a registry of reserved names. 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 computed over the assertion text, beginning with the first field (including the field identifier string), up to (but not including) the Signature field identifier. The newline preceding the signature field identifier is the last character included in signature calculation. The signature is always the last field in a KeyNote assertion. Text following this field is not considered part of the assertion.

The algorithms for computing and verifying signatures must be configured into each KeyNote implementation and are defined and documented separately.

Note that all signatures used in examples in this document are fictitious and generally much shorter than would be required for security in practice.

## 5. Query Evaluation Semantics

The KeyNote compliance checker finds and returns the Policy Compliance Value of queries, as defined in Section 5.3, below.

### 5.1 Query Parameters

A KeyNote query has four parameters:

- \* The identifier of the principal(s) requesting the action.
- \* The action attribute set describing the action.
- \* The set of compliance values of interest to the application, ordered from `_MIN_TRUST` to `_MAX_TRUST`
- \* The policy and credential assertions that should be included in the evaluation.

The mechanism for passing these parameters to the KeyNote evaluator is application dependent. In particular, an evaluator might provide for some parameters to be passed explicitly, while others are looked up externally (e.g., credentials might be looked up in a network-based distribution system), while still others might be requested from the application as needed by the evaluator, through a 'callback' mechanism (e.g., for attribute values that represent values from among a very large namespace).

#### 5.1.1 Action Requester

At least one Principal must be identified in each query as the 'requester' of the action. Actions may be requested by several principals, each considered to have individually requested it. This allows policies that require multiple authorizations, e.g., 'two person control'. The set of authorizing principals is made available in the special attribute `"_ACTION_AUTHORIZERS"`; if several principals are authorizers, their identifiers are separated with commas.

### 5.1.2 Ordered Compliance Value Set

The set of compliance values of interest to an application (and their relative ranking to one another) is determined by the invoking application and passed to the KeyNote evaluator as a parameter of the query. In many applications, this will be Boolean, e.g., the ordered sets {FALSE, TRUE} or {REJECT, APPROVE}. Other applications may require a range of possible values, e.g., {No\_Access, Limited\_Access, Full\_Access}. Note that applications should include in this set only compliance value names that are actually returned by the assertions.

The lowest-order and highest-order compliance value strings given in the query are available in the special attributes named "\_MIN\_TRUST" and "\_MAX\_TRUST", respectively. The complete set of query compliance values is made available in ascending order (from \_MIN\_TRUST to \_MAX\_TRUST) in the special attribute named "\_VALUES". Values are separated with commas; applications that use assertions that make use of the \_VALUES attribute should therefore avoid the use of compliance value strings that themselves contain commas.

## 5.2 Principal Identifier Normalization

Principal identifier comparisons among Cryptographic Principal Identifiers (that represent keys) in the Authorizer and Licensees fields or in an action's direct authorizers are performed after normalizing them by conversion to a canonical form.

Every cryptographic algorithm used in KeyNote defines a method for converting keys to their canonical form and that specifies how the comparison for equality of two keys is performed. If the algorithm named in the identifier is unknown to KeyNote, the identifier is treated as opaque.

Opaque identifiers are compared as case-sensitive strings.

Notice that use of opaque identifiers in the Authorizer field requires that the assertion's integrity be locally trusted (since it cannot be cryptographically verified by the compliance checker).

## 5.3 Policy Compliance Value Calculation

The Policy Compliance Value of a query is the Principal Compliance Value of the principal named "POLICY". This value is defined as follows:

### 5.3.1 Principal Compliance Value

The Compliance Value of a principal <X> is the highest order (maximum) of:

- the Direct Authorization Value of principal <X>; and
- the Assertion Compliance Values of all assertions identifying <X> in the Authorizer field.

### 5.3.2 Direct Authorization Value

The Direct Authorization Value of a principal <X> is `_MAX_TRUST` if <X> is listed in the query as an authorizer of the action. Otherwise, the Direct Authorization Value of <X> is `_MIN_TRUST`.

### 5.3.3 Assertion Compliance Value

The Assertion Compliance Value of an assertion is the lowest order (minimum) of the assertion's Conditions Compliance Value and its Licensee Compliance Value.

### 5.3.4 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, as defined Section 5.1.

If an assertion's Conditions field is missing entirely, its Conditions Compliance Value is considered to be the `_MAX_TRUST` value, as defined in Section 5.1.

The set of successful test clause values is calculated as follows:

Recall from the grammar of section 4.6.5 that 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 ordered set 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-order 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-order possible 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, \_2, ..., \_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.

### 5.3.5 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.

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 Section 4.6.4) 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.

#### 5.4 Assertion Management

Assertions may be either signed or unsigned. Only signed assertions should be used as credentials or transmitted or stored on untrusted media. Unsigned assertions should be used only to specify policy and for assertions whose integrity has already been verified as conforming to local policy by some mechanism external to the KeyNote system itself (e.g., X.509 certificates converted to KeyNote assertions by a trusted conversion program).

Implementations that permit signed credentials to be verified by the KeyNote compliance checker generally provide two 'channels' through which applications can make assertions available. Unsigned, locally-trusted assertions are provided over a 'trusted' interface, while signed credentials are provided over an 'untrusted' interface. The KeyNote compliance checker verifies correct signatures for all assertions submitted over the untrusted interface. The integrity of KeyNote evaluation requires that only assertions trusted as reflecting local policy are submitted to KeyNote via the trusted interface.



Note that applications that use KeyNote exclusively as a local policy specification mechanism need use only trusted assertions. Other applications might need only a small number of infrequently changed trusted assertions to 'bootstrap' a policy whose details are specified in signed credentials issued by others and submitted over the untrusted interface.

## 5.5 Implementation Issues

Informally, the semantics of KeyNote evaluation can be thought of as involving the construction a directed graph of KeyNote assertions rooted at a POLICY assertion that connects with at least one of the principals that requested the action.

Delegation of some authorization from principal <A> to a set of principals <B> is expressed as an assertion with principal <A> given in the Authorizer field, principal set <B> given in the Licensees field, and the authorization to be delegated encoded in the Conditions field. How the expression digraph is constructed is implementation-dependent and implementations may use different algorithms for optimizing the graph's construction. Some implementations might use a 'bottom up' traversal starting at the principals that requested the action, others might follow a 'top down' approach starting at the POLICY assertions, and still others might employ other heuristics entirely.

Implementations are encouraged to employ mechanisms for recording exceptions (such as division by zero or syntax error), and reporting them to the invoking application if requested. Such mechanisms are outside the scope of this document.

## 6. Examples

In this section, we give examples of KeyNote assertions that might be used in hypothetical applications. These examples are intended primarily to illustrate features of KeyNote assertion syntax and semantics, and do not necessarily represent the best way to integrate KeyNote into applications.

In the interest of readability, we use much shorter keys than would ordinarily be used in practice. Note that the Signature fields in these examples do not represent the result of any real signature calculation.

## 1. TRADITIONAL CA / EMAIL

- A. A policy unconditionally authorizing RSA key abc123 for all actions. This essentially defers the ability to specify policy to the holder of the secret key corresponding to abc123:

```
Authorizer: "POLICY"
Licensees: "RSA:abc123"
```

- B. A credential assertion in which RSA Key abc123 trusts either RSA key 4401ff92 (called 'Alice') or DSA key d1234f (called 'Bob') to perform actions in which the "app\_domain" is "RFC822-EMAIL", where the "address" matches the regular expression "^.\*@keynote\.research\.att\.com\$". In other words, abc123 trusts Alice and Bob as certification authorities for the keynote.research.att.com domain.

```
KeyNote-Version: 2
Local-Constants: Alice="DSA:4401ff92"  # Alice's key
                  Bob="RSA:d1234f"      # Bob's key
Authorizer: "RSA:abc123"
Licensees: Alice || Bob
Conditions: (app_domain == "RFC822-EMAIL") &&
            (address ~= # only applies to one domain
             "^.*@keynote\\.research\\.att\\.com$");
Signature: "RSA-SHA1:213354f9"
```

- C. A certificate credential for a specific user whose email address is mab@keynote.research.att.com and whose name, if present, must be "M. Blaze". The credential was issued by the 'Alice' authority (whose key is certified in Example B above):

```
KeyNote-Version: 2
Authorizer: "DSA:4401ff92"  # the Alice CA
Licensees: "DSA:12340987"   # mab's key
Conditions: ((app_domain == "RFC822-EMAIL") &&
            (name == "M. Blaze" || name == "")) &&
            (address == "mab@keynote.research.att.com"));
Signature: "DSA-SHA1:ab23487"
```

- D. Another certificate credential for a specific user, also issued by the 'Alice' authority. This example allows three different keys to sign as jf@keynote.research.att.com (each for a different cryptographic algorithm). This is, in effect, three credentials in one:

```

KeyNote-Version: "2"
Authorizer: "DSA:4401ff92"      # the Alice CA
Licensees: "DSA:abc991" ||      # jf's DSA key
           "RSA:cde773" ||      # jf's RSA key
           "BFIK:fd091a"        # jf's BFIK key
Conditions: ((app_domain == "RFC822-EMAIL") &&
             (name == "J. Feigenbaum" || name == "") &&
             (address == "jf@keynote.research.att.com"));
Signature: "DSA-SHA1:8912aa"

```

Observe that under policy A and credentials B, C and D, the following action attribute sets are accepted (they return `_MAX_TRUST`):

```

_ACTION_AUTHORIZERS = "dsa:12340987"
app_domain = "RFC822-EMAIL"
address = "mab@keynote.research.att.com"
and
_ACTION_AUTHORIZERS = "dsa:12340987"
app_domain = "RFC822-EMAIL"
address = "mab@keynote.research.att.com"
name = "M. Blaze"

```

while the following are not accepted (they return `_MIN_TRUST`):

```

_ACTION_AUTHORIZERS = "dsa:12340987"
app_domain = "RFC822-EMAIL"
address = "angelos@dsl.cis.upenn.edu"
and
_ACTION_AUTHORIZERS = "dsa:abc991"
app_domain = "RFC822-EMAIL"
address = "mab@keynote.research.att.com"
name = "M. Blaze"
and
_ACTION_AUTHORIZERS = "dsa:12340987"
app_domain = "RFC822-EMAIL"
address = "mab@keynote.research.att.com"
name = "J. Feigenbaum"

```

## 2. WORKFLOW/ELECTRONIC COMMERCE

- E. A policy that delegates authority for the "SPEND" application domain to RSA key dab212 when the amount given in the "dollars" attribute is less than 10000.

```

Authorizer: "POLICY"
Licensees: "RSA:dab212" # the CFO's key
Conditions: (app_domain=="SPEND") && (@dollars < 10000);

```

- F. RSA key dab212 delegates authorization to any two signers, from a list, one of which must be DSA key feed1234 in the "SPEND" application when @dollars < 7500. If the amount in @dollars is 2500 or greater, the request is approved but logged.

```

KeyNote-Version: 2
Comment: This credential specifies a spending policy
Authorizer: "RSA:dab212" # the CFO
Licensees: "DSA:feed1234" && # The vice president
           ("RSA:abc123" || # middle manager #1
            "DSA:bcd987" || # middle manager #2
            "DSA:cde333" || # middle manager #3
            "DSA:def975" || # middle manager #4
            "DSA:978add") # middle manager #5
Conditions: (app_domain=="SPEND") # note nested clauses
           -> { (@(dollars) < 2500)
                -> _MAX_TRUST;
                (@(dollars) < 7500)
                -> "ApproveAndLog";
              };
Signature: "RSA-SHA1:9867a1"

```

- G. According to this policy, any two signers from the list of managers will do if @(dollars) < 1000:

```

KeyNote-Version: 2
Authorizer: "POLICY"
Licensees: 2-of("DSA:feed1234", # The VP
               "RSA:abc123", # Middle management clones
               "DSA:bcd987",
               "DSA:cde333",
               "DSA:def975",
               "DSA:978add")
Conditions: (app_domain=="SPEND") &&
           (@(dollars) < 1000);

```

- H. A credential from dab212 with a similar policy, but only one signer is required if  $@(dollars) < 500$ . A log entry is made if the amount is at least 100.

```

KeyNote-Version: 2
Comment: This one credential is equivalent to six separate
        credentials, one for each VP and middle manager.
        Individually, they can spend up to $500, but if
        it's $100 or more, we log it.
Authorizer: "RSA:dab212"           # From the CFO
Licensees: "DSA:feed1234" ||      # The VP
           "RSA:abc123" ||         # The middle management clones
           "DSA:bcd987" ||
           "DSA:cde333" ||
           "DSA:def975" ||
           "DSA:978add"
Conditions: (app_domain="SPEND") # nested clauses
            -> { (@(dollars) < 100) -> _MAX_TRUST;
                (@(dollars) < 500) -> "ApproveAndLog";
            };
Signature: "RSA-SHA1:186123"

```

Assume a query in which the ordered set of Compliance Values is {"Reject", "ApproveAndLog", "Approve"}. Under policies E and G, and credentials F and H, the Policy Compliance Value is "Approve" ( $_MAX\_TRUST$ ) when:

```

_ACTION_AUTHORIZERS = "DSA:978add"
app_domain = "SPEND"
dollars = "45"
unmentioned_attribute = "whatever"
and
_ACTION_AUTHORIZERS = "RSA:abc123,DSA:cde333"
app_domain = "SPEND"
dollars = "550"

```

The following return "ApproveAndLog":

```

_ACTION_AUTHORIZERS = "DSA:feed1234,DSA:cde333"
app_domain = "SPEND"
dollars = "5500"
and
_ACTION_AUTHORIZERS = "DSA:cde333"
app_domain = "SPEND"
dollars = "150"

```

However, the following return "Reject" (`_MIN_TRUST`):

```
_ACTION_AUTHORIZERS = "DSA:def975"
app_domain = "SPEND"
dollars = "550"
and
_ACTION_AUTHORIZERS = "DSA:cde333,DSA:978add"
app_domain = "SPEND"
dollars = "5500"
```

## 7. Trust-Management Architecture

KeyNote provides a simple mechanism for describing security policy and representing credentials. It differs from traditional certification systems in that the security model is based on binding keys to predicates that describe what the key is authorized by policy to do, rather than on resolving names. The infrastructure and architecture to support a KeyNote system is therefore rather different from that required for a name-based certification scheme. The KeyNote trust-management architecture is based on that of PolicyMaker [BFL96,BFS98].

It is important to understand the separation between the responsibilities of the KeyNote system and those of the application and other support infrastructure. A KeyNote compliance checker will determine, based on policy and credential assertions, whether a proposed action is permitted according to policy. The usefulness of KeyNote output as a policy enforcement mechanism depends on a number of factors:

- \* The action attributes and the assignment of their values must reflect accurately the security requirements of the application. Identifying the attributes to include in the action attribute set is perhaps the most important task in integrating KeyNote into new applications.
- \* The policy of the application must be correct and well-formed. In particular, trust must be deferred only to principals that should, in fact, be trusted by the application.
- \* The application itself must be trustworthy. KeyNote does not directly enforce policy; it only provides advice to the applications that call it. In other words, KeyNote assumes that the application itself is trusted and that the policy assertions it specifies are correct. Nothing prevents an application from submitting misleading or incorrect assertions to KeyNote or from ignoring KeyNote altogether.

It is also up to the application (or some service outside KeyNote) to select the appropriate credentials and policy assertions with which to run a particular query. Note, however, that even if inappropriate credentials are provided to KeyNote, this cannot result in the approval of an illegal action (as long as the policy assertions are correct and the the action attribute set itself is correctly passed to KeyNote).

KeyNote is monotonic; adding an assertion to a query can never result in a query's having a lower compliance value that it would have had without the assertion. Omitting credentials may, of course, result in legal actions being disallowed. Selecting appropriate credentials (e.g., from a distributed database or 'key server') is outside the scope of the KeyNote language and may properly be handled by a remote client making a request, by the local application receiving the request, or by a network-based service, depending on the application.

In addition, KeyNote does not itself provide credential revocation services, although credentials can be written to expire after some date by including a date test in the predicate. Applications that require credential revocation can use KeyNote to help specify and implement revocation policies. A future document will address expiration and revocation services in KeyNote.

Because KeyNote is designed to support a variety of applications, several different application interfaces to a KeyNote implementation are possible. In its simplest form, a KeyNote compliance checker would exist as a stand-alone application, with other applications calling it as needed. KeyNote might also be implemented as a library to which applications are linked. Finally, a KeyNote implementation might run as a local trusted service, with local applications communicating their queries via some interprocess communication mechanism.

## 8. Security Considerations

Trust management is itself a security service. Bugs in or incorrect use of a KeyNote compliance checker implementation could have security implications for any applications in which it is used.

## 9. IANA Considerations

This document contains three identifiers to be maintained by the IANA. This section explains the criteria to be used by the IANA to assign additional identifiers in each of these lists.

### 9.1 app\_domain Identifiers

The only thing required of IANA on allocation of these identifiers is that they be unique strings. These strings are case-sensitive for KeyNote purposes, however it is strongly recommended that IANA assign different capitalizations of the same string only to the same organization.

### 9.2 Public Key Format Identifiers

These strings uniquely identify a public key algorithm as used in the KeyNote system for representing keys. Requests for assignment of new identifiers must be accompanied by an RFC-style document that describes the details of this encoding. Example strings are "rsa-hex:" and "dsa-base64:". These strings are case-insensitive.

### 9.3 Signature Algorithm Identifiers

These strings uniquely identify a public key algorithm as used in the KeyNote system for representing public key signatures. Requests for assignment of new identifiers must be accompanied by an RFC-style document that describes the details of this encoding. Example strings are "sig-rsa-md5-hex:" and "sig-dsa-sha1-base64:". Note that all such strings must begin with the prefix "sig-". These strings are case-insensitive.



## A. Acknowledgments

We thank Lorrie Faith Cranor (AT&T Labs - Research) and Jonathan M. Smith (University of Pennsylvania) for their suggestions and comments on earlier versions of this document.

## B. Full BNF (alphabetical order)

```

<ALGORITHM>:: {see section 4.4.2} ;

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

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

<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} ;

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

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

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

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

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

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

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

<DerefAttribute>:: <AttributeID> ;

<ENCODEDBITS>:: {see section 4.4.2} ;

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

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

```

```

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

<IDString>:: <ALGORITHM> ":" <ENCODEDBITS> ;

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

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

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

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

<KeyID>:: <StrEx> ;

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

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

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

<OpaqueID>:: <StrEx> ;

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

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

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

<RegExpr>:: {POSIX 1003.2 Regular Expression}

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

<Signature>:: <StrEx> ;

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

```

```

<StrEx>:: <StrEx> "." <StrEx> | <StringLiteral> | "(" <StrEx> ")"
          | <DerefAttribute> | "$" <StrEx> ;

<StringLiteral>:: {see section 4.3.1} ;

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

<Test>:: <RelExpr> ;

<Value>:: <StrEx> ;

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

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

```

## References

- [BFL96] M. Blaze, J. Feigenbaum, J. Lacy. Decentralized Trust Management. Proceedings of the 17th IEEE Symp. on Security and Privacy. pp 164-173. IEEE Computer Society, 1996. Available at  
<ftp://ftp.research.att.com/dist/mab/policymaker.ps>
- [BFS98] M. Blaze, J. Feigenbaum, M. Strauss. Compliance-Checking in the PolicyMaker Trust-Management System. Proc. 2nd Financial Crypto Conference. Anguilla 1998. LNCS #1465, pp 251-265, Springer-Verlag, 1998. Available at  
<ftp://ftp.research.att.com/dist/mab/pmcomply.ps>
- [Bla99] M. Blaze, J. Feigenbaum, J. Ioannidis, A. Keromytis. The Role of Trust Management in Distributed System Security. Chapter in Secure Internet Programming: Security Issues for Mobile and Distributed Objects (Vitek and Jensen, eds.). Springer-Verlag, 1999. Available at  
<ftp://ftp.research.att.com/dist/mab/trustmgt.ps>.
- [Cro82] Crocker, D., "Standard for the Format of ARPA Internet Text Messages", STD 11, RFC 822, August 1982.
- [DSA94] Digital Signature Standard. FIPS-186. National Institute of Standards, U.S. Department of Commerce. May 1994.
- [PKCS1] PKCS #1: RSA Encryption Standard, Version 1.5. RSA Laboratories. November 1993.

[RSA78] R. L. Rivest, A. Shamir, L. M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM, v21n2. pp 120-126. February 1978.

#### Authors' Addresses

Comments about this document should be discussed on the keynote-users mailing list hosted at nsa.research.att.com. To subscribe, send an email message containing the single line  
subscribe keynote-users  
in the message body to <majordomo@nsa.research.att.com>.

Questions about this document can also be directed to the authors as a group at the keynote@research.att.com alias, or to the individual authors at:

Matt Blaze  
AT&T Labs - Research  
180 Park Avenue  
Florham Park, New Jersey 07932-0971

Email: mab@research.att.com

Joan Feigenbaum  
AT&T Labs - Research  
180 Park Avenue  
Florham Park, New Jersey 07932-0971

Email: jf@research.att.com

John Ioannidis  
AT&T Labs - Research  
180 Park Avenue  
Florham Park, New Jersey 07932-0971

Email: ji@research.att.com

Angelos D. Keromytis  
Distributed Systems Lab  
CIS Department, University of Pennsylvania  
200 S. 33rd Street  
Philadelphia, Pennsylvania 19104-6389

Email: angelos@dsl.cis.upenn.edu

## Full Copyright Statement

Copyright (C) The Internet Society (1999). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

## Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

