FIPS PUB XXX

FEDERAL INFORMATION
PROCESSING STANDARDS PUBLICATION

DRAFT 1994 May 23 DRAFT

U.S. DEPARTMENT OF COMMERCE / National Institute of Standards and Technology

CRYPTOGRAPHIC SERVICE CALLS

(DRAFT)

CATEGORY: COMPUTER SECURITY
SUBCATEGORY: CRYPTOGRAPHY

U.S. DEPARTMENT OF COMMERCE, Ronald H. Brown, *Secretary*
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY,
Dr. Arati Prabhakar, *Director*

Foreword

The Federal Information Processing Standards Publication Series of the National Institute of Standards and Technology (NIST) is the official series of publications relating to standards and guidelines adopted and promulgated under the provisions of Section 111(d) of the Federal Property and Administrative Services Act of 1949 as amended by the Computer Security Act of 1987, Public Law 100-235. These mandates have given the Secretary of Commerce and NIST important responsibilities for improving the utilization and management of computer and related telecommunications systems in the Federal Government. The NIST, through its Computer Systems Laboratory, provides leadership, technical guidance, and coordination of Government efforts in the development of standards and guidelines in these areas.

Comments concerning Federal Information Processing Standards Publications are welcomed and should be addressed to the Director, Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899.

James H. Burrows, *Director*
Computer Systems Laboratory

Abstract

Cryptography has increasing been considered or used in the Federal government to protect sensitive information and provide adequate security in its computers and telecommunication systems. This standard specifies a set of generic cryptographic service calls for application programs to request common cryptographic functions from a cryptographic module which provides these cryptographic capabilities. The service calls address both the secret-key based and the public-key based cryptographic algorithms. Cryptographic functions specified in this document include: message encryption and decryption, message authentication, digital signature generation and verification, key management, and user authentication. The standard interface is aimed to promote interoperability among different cryptographic implementations.

Key words: computer security, cryptography, cryptographic Application Program Interface (cryptographic API), Federal Information Processing Standard (FIPS).

Proposed
Federal Information
Processing Standards Publication XXX

1994 May 23

Announcing the Standard for

CRYPTOGRAPHIC SERVICE CALLS

Federal Information Processing Standards Publications (FIPS PUBS) are issued by the National Institute of Standards and Technology (NIST) after approval by the Secretary of Commerce pursuant to Section 111(d) of the Federal Property and Administrative Services Act of 1949 as amended by the Computer Security Act of 1987, Public Law 100-235.

1. Name of Standard. Cryptographic Service Calls (FIPS PUB XXX).

2. Category of Standard. Computer Security, Cryptography.

3. Explanation. This standard specifies a set of generic cryptographic service calls, or applications program interface (API), for application programs to interface with a cryptographic module for requesting cryptographic functions. The service calls specify the interface for common cryptographic functions such as message encryption and decryption, message authentication, digital signature generation and verification, key management, and user authentication. Cryptographic algorithms that are supported include both secret-key based and public-key based algorithms. In this standard, the terms *cryptographic service calls* and *cryptographic APIs* can be used interchangeably.

4. Approving Authority. Secretary of Commerce.

5. Maintenance Agency. Department of Commerce, National Institute of Standards and Technology, (Computer Systems Laboratory).

6. Cross Index.

a. FIPS PUB 46-2, Data Encryption Standard.
b. FIPS PUB 74, Guidelines for Implementing and Using the NBS Data Encryption Standard.
c. FIPS PUB 81, DES Modes of Operation.
d. FIPS PUB 113, Computer Data Authentication.
e. FIPS PUB 171, Key Management Using ANSI X9.17.

f. FIPS PUB 180, Secure Hash Standard.
g. FIPS PUB XXX, Digital Signature Standard.
h. FIPS PUB 185, Escrowed Encryption Standard.
i. Special Publication 800-2, Public Key Cryptography.
j. Federal Information Resources Management Regulations (FIRMR) subpart 201.20.303, Standards, and subpart 201.39.1002, Federal Standards.

Other NIST publications may be applicable to the implementation and use of this standard. A list (NIST Publications List 91) of currently available computer security publications, including ordering information, can be obtained from NIST.

7. Objectives. A standard cryptographic interface will facilitate interoperability among different cryptographic implementations. Specifically, a standard set of cryptographic service calls provides the following advantages:

a. Application programmers will need to learn only one set of cryptographic service calls for multiple cryptographic applications.

b. Cryptographic modules from different vendors, which conform to this interface standard, may be interfaced to a given application without requiring modification to the application program.

c. Contracts for additional cryptographic modules would not have to be sole sourced because multiple vendors would offer the standard service calls.

d. Vendors could build cryptographic modules which would interface to a wide variety of applications.

8. Applicability. This standard is applicable to all Federal departments and agencies that use cryptographic-based security systems for the protection of unclassified information that is not subject to Section 2315 of Title 10, U.S. Code, or Section 3502(2) of Title 44, U.S. Code. The standard shall be used by all Federal departments and agencies in designing, acquiring and implementing cryptographic services where a cryptographic interface is to be provided. Not all of the service calls specified in this standard need to be used in its entirety by an application. The specific service calls that shall be used depend on the security requirements for the particular application and environment in which the system is to be utilized. Private and commercial organizations are encouraged to adopt and use this standard in order to facilitate interoperability among different cryptographic products.

9. Applications. The standard may be used in any application which uses cryptography to provide any of the following cryptographic functions: message encryption/decryption, message authentication, digital signature generation and verification, and key management. Not all the service calls specified in this standard need to be used by an application. An application can make use of additional service calls not available in this standard.

10. Specifications. Federal Information Processing Standard (FIPS) XXX, *Cryptographic Service Calls*.

11. Implementations. Though this document specifies a standard interface for requesting cryptographic functions, the standard, however, does not mandate a specific implementation of these cryptographic functions other than what are explicitly specified in the document. The cryptographic functions may in fact be implemented in software, firmware, hardware, or any combination thereof. However, there may be other standards that are applicable to the implementation of specific cryptographic functions. For specific requirements, the individual standard shall be referred to. Conformance to this standard requires that the cryptographic service calls used by an application provide exactly the same name and letter case for the service calls and their parameters as specified in the standard. In the rare case where the standard naming and specification of the service calls and parameters may violate certain rules of a particular programming language in use, the exception should be noted and the selected naming and case specification should match the standard as much as possible.

12. Export Control. Certain cryptographic devices and technical data regarding them are deemed to be defense articles (i.e., inherently military in character) and are subject to Federal government export controls as specified in Title 22, Code of Federal Regulations, Parts 120-128. Some exports of cryptographic modules conforming to this standard and technical data regarding them must comply with these Federal regulations and be licensed by the U.S. Department of State. Other exports of cryptographic modules conforming to this standard and technical data regarding them fall under the licensing authority of the Bureau of Export Administration of the U.S. Department of Commerce. The Department of Commerce is responsible for licensing cryptographic devices used for authentication, access control, proprietary software, automatic teller machines (ATMs), and certain devices used in other equipment and software. For advice concerning which agency has licensing authority for a particular cryptographic device, please contact the respective agencies.

13. Implementation Schedule. This standard becomes effective six months after publication of a notice in the Federal Register of its approval by the Secretary of Commerce.

14. Qualifications. While this standard specifies a standard interface for application programs to request cryptographic functions from a cryptographic module, conformance to this standard does not assure that a particular cryptographic module or implementation is secure. Security requirements for a cryptographic module are addressed in FIPS 140-1. The responsible authority in each agency or department shall assure that the overall system provides an acceptable level of security.

15. Waiver Procedure. Under certain exceptional circumstances, the heads of Federal departments and agencies may approve waivers to Federal Information Processing Standards (FIPS). The head of such agency may redelegate such authority only to a senior official designated pursuant to Section 3506(b) of Title 44, U.S. Code. Waivers shall be granted only when:

a. Compliance with a standard would adversely affect the accomplishment of the mission of an operator of a Federal computer system, or

b. cause a major adverse financial impact on the operator which is not offset by Government-wide savings.

Agency heads may act upon a written waiver request containing the information detailed above. Agency heads may also act without a written waiver request when they determine that conditions for meeting the standard cannot be met. Agency heads may approve waivers only by a written decision which explains the basis on which the agency head made the required finding(s). A copy of each such decision, with procurement sensitive or classified portions clearly identified, shall be sent to: National Institute of Standards and Technology; ATTN: FIPS Waiver Decisions, Technology Building, Room B-154; Gaithersburg, MD 20899.

In addition, notice of each waiver granted and each delegation of authority to approve waivers shall be sent promptly to the Committee on Government Operations of the House of Representatives and the Committee on Government Affairs of the Senate and shall be published promptly in the Federal Register.

When the determination on a waiver applies to the procurement of equipment and/or services, a notice of the waiver determination must be published in the Commerce Business Daily as a part of the notice of solicitation for offers of an acquisition or, if the waiver determination is made after that notice is published, by amendment to such notice.

A copy of the waiver, any supporting documents, the document approving the waiver and any supporting and accompanying documents, with such deletions as the agency is authorized and decides to make under Section 552(b) of Title 5, U.S. Code, shall be part of the procurement documentation and retained by the agency.

16. Where to obtain copies. Copies of this publication are available for sale by the National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. When ordering, refer to Federal Information Processing Standards Publication XXX (FIPS PUB XXX), and title. When microfiche is desired, this should be specified. Payment may be made by check, money order, credit card, or deposit account.

TABLE OF CONTENTS

CRYPTOGRAPHIC SERVICE CALLS

Proposed
Federal Information
Processing Standards Publication XXX

1994 May 23

Specifications for the

CRYPTOGRAPHIC SERVICE CALLS

1 INTRODUCTION

1.1 Background

The prevalence of computers and computer networks has greatly influenced the way people work; unfortunately, the lack of properly protected computer systems and networks has also contributed to a notable increase in computer related crimes. Without proper protection, the information stored in a computer or travelling in a network can easily be accessed, intercepted or altered. Depending on the sensitivity of the information and the type of tampering, the potential damage can be significant.

Cryptography has been known for decades and used often during wartime to protect sensitive or secret information from unauthorized personnel while the information was delivered via unsecured channels. Using encryption, a message is transformed into a form unreadable by anyone without a secret decryption key. Thus, encryption can protect the privacy of information traveling in unsecured networks. Cryptography may also be used to protect the integrity of information by a process called message authentication and verification [1]. The process involves the calculation of a computer checksum based on the message to be sent, the checksum is sent along with the message to a recipient, who may recompute the checksum and verify that the message was not modified in transit.

To meet the increasing demand for information security, many vendors have designed and marketed cryptographic products. Though capabilities vary among these products, most of them provide a set of basic cryptographic functions, including message encryption and decryption, message authentication and verification, and key management functions. Two branches of cryptographic technologies prevail today: secret key cryptography and public key cryptography. Both techniques will be discussed in later sections. Regardless of the technique used, each has some keying information to keep and manage, thus each needs a properly implemented key management system.

To protect the secret information such as users' secret keys from unwanted disclosure, cryptographic functions are frequently implemented in a secured module. A *cryptographic module* (CM) is a set of hardware, firmware or software, or some combination thereof, that implements cryptographic logic and/or processes [2]. The FIPS 140-1 [2] addresses cryptographic modules and their security requirements in detail. A cryptographic module may also be used to generate cryptographic keys and to encrypt keys for storage. It is obvious that without proper protection to the CM, any security service provided by it is totally useless.

To meet the security requirements of a CM and its operation, security applications usually request cryptographic functions from the CM through a controlled interface. The cryptographic interface has not been standardized in the past. The purpose of this standard is to define a generic set of *cryptographic service calls* to support basic cryptographic functions commonly found in a CM. A standard set of cryptographic service calls provides the following advantages:

a. Application programmers will need to learn only one set of cryptographic service calls for multiple cryptographic applications.

b. Cryptographic modules from different vendors, which conform to this interface standard, may be interfaced to a given application without requiring modification to the application program.

c. Contracts for additional cryptographic modules would not have to be sole sourced because multiple vendors would offer the standard service calls.

d. Vendors could build cryptographic modules which would interface to a wide variety of applications.

In short, a standard cryptographic interface will permit interoperability among different cryptographic implementations. The standard shall be used when Federal agencies desire to specify an application layer cryptographic interface in government contracts. Private and commercial organizations are encouraged to adopt and use this standard when cryptographic interfaces are to be provided for their cryptographic products.

It should be noted that the aim of this document is to standardize the cryptographic interface rather than the specific implementation of cryptographic functions in a CM. Although each cryptographic service call is expected to provide a certain function to the calling program and through a defined interface, different implementations of a service call are possible and permitted. A cryptographic module may also provide additional cryptographic capabilities not covered in this standard. The set of cryptographic service calls can be integrated, in full or in subset, into any security application.

1.2 Overview of Secret-Key and Public-Key Cryptography

Since a major part of this document specifies the cryptographic service calls for secret-key and public-key cryptosystems, an overview of these cryptosystems will be helpful. However, to fully understand these service calls, some familiarity with both cryptosystems is required. Interested readers may wish to read [3] and [4] to learn more about the two prime cryptographic technologies.

In secret-key cryptography, a secret key is established and shared between two individuals or parties and the same key is used to encrypt or decrypt messages, therefore, it is also referred to as symmetric cryptography. If the two parties are in different physical locations, they must trust a courier, or some transmission system to establish the initial key and trust this third-party not to disclose the secret key they are communicating. The generation, transmission, and storage of keys is called key management. Ensuring that key storage, exchange of new keys and destruction of old keys are performed securely often creates complex key management requirements for secret key cryptography. The ANSI X9.17 Financial Institution Key Management (Wholesale) Standard prescribes a uniform process for the protection and exchange of cryptographic keys for authentication and encryption in the financial community [5].

In a public-key system, a user makes use of a pair of keys: a public key and a private key. The two keys are uniquely related so that the public key of a user can be made public without revealing any information about the private key. The private key of a user is usable only by its owner. Because the value of the private key is not shared, public-key cryptography is often considered to be more secure than secret-key cryptography. It is the specific feature that a private key is never shared with another party that permits the unique signing capability. With public-key cryptography, no single key is used for both encryption and decryption. Thus, public-key cryptography is also known as asymmetric cryptography.

It is beyond the scope of this document to describe how public-key cryptography works, interested readers are referred to [6] for details. Since a user's public key is made public, certain control is necessary so that a user's public key can not be altered. The application of public-key cryptography thus requires an authentication framework which binds users' public keys and users' identities. A public-key *certificate* is a certified proof of such a binding vouched by a trusted third-party called Certification Authority (CA). The use of a CA alleviates the responsibility of individual users to verify directly the binding of a user's public key to the user. Reference [7] addresses issues involved for managing public-key certificates.

1.3 Cryptographic Database

Before the cryptographic service calls are presented, it is necessary to discuss cryptographic databases that are essential for supporting cryptographic operations. These databases are described as follows.

A. A user authentication database (UDATABASE) must exist. A user must be authenticated before making any service request to the CM. Once authenticated, a user is considered "logged in" to the CM and a connection is established between the user's application process and the CM. If multiple users can log in to the CM simultaneously and share its resources, it is the host operating system's and CM's responsibilities to maintain the separation of service calls among simultaneous connections. It is therefore assumed that the CM knows the identity of the user executing any CM service call until the user specifically logs out of the CM.

   Each element in UDATABASE should contain at least the following fields: user id, user authenticator, user type, and user authorization vector. *User id* is the user's login identifier. The user authenticator can be a password, a biometrics template, or anything that can be used to authenticate a user. Access to the CM may be controlled through different access privileges. The user having the highest authorization privilege is the cryptographic officer (CO). The field *user type* is used to indicate whether a user is a CO or a regular user. The CO assigns specific cryptographic service calls that a user can access in the user authorization vector. When a CM is used for the first time, A cryptographic officer should initialize the CM and the UDATABASE would then contain an entry for the CO.

B. For the secret key cryptography, a secure secret key database (SKEYDB) must exist to store the secret keys. Each key in SKEYDB must be identifiable either by a name or by a reference number. Keys owned by an individual user are further required to have unique identifiers. Thus, a key can be uniquely identified by the combination of a user id and a key id. This document assumes that keys in SKEYDB are referenced by names.

   Proper keying facilities must be provided to control the lifecycle of a key and ensure that replacement keys are brought into operation securely and old keys are safely destroyed. SKEYDB may contain these fields: user id, key id, key value, key type, and key counter if applicable. Generally speaking, storage space is more limited in a CM than in a host computer. Therefore, SKEYDB is more likely to reside in the host, though this is not a requirement. Keys stored outside the CM must be protected by encryption. It is possible and may be desirable to combine UDATABASE and SKEYDB into one single database, which is a design issue to be determined by the implementor.

   For the operation of cryptographic functions, keys must be loaded into the proper registers of the CM before cryptographic functions can take place. These registers are CM-dependent and are not to be confused with the generic secure key database (SKEYDB).

The retrieval of keys from SKEYDB to the CM registers is handled by the cryptographic module rather than by the application programs, therefore, no cryptographic service call is defined for key retrieval for the CM. For easy referencing, let the registers of a CM be called CMKEYREG. Depending on the type of CM used, the storage of keys in CMKEYREG may be temporary whereas the storage of keys in SKEYDB is more permanent, that is until a key is specifically removed. Keys can be loaded to or removed from SKEYDB by cryptographic service calls.

C. For public key cryptography, a user's public and private key pair is normally generated and stored in the individual user's CM. A user's public key can be stored in a public directory, however, it must be encapsulated in a public key certificate in order for other users to verify the validity of the public key.

A user may wish to create a local key database in order to store the user's key pair and the associated public key certificate. A user may also cache the public keys or certificates of other users that he frequently communicates with. The keying information may be loaded into and removed from the key storage by cryptographic service calls. To facilitate parameter description of the public-key based service calls, pertinent data structures are introduced in Section 2.4 before the public key cryptographic service calls are presented.

It is assumed that each key pair and the associated public key certificate will be stored as a single record and the key pair will share a unique identifier which is represented as *pubkeyid* in this document. The keying information is retrievable and removable as a single record as implied in the data structure *PubKeyRec* (Section 2.4). If a user owns multiple public-private key pairs, each key pair must have a unique identifier. A public key may consist of several components. Depending on the specific algorithm used, some of the components may be shared by multiple users. In this document, no assumption is made that the public key components p, q, and g of the Digital Signature Algorithm (DSA) must be shared globally.

1.4 Organization of the Document

Section 2 presents the cryptographic service calls. Section 2.1 lists all of the cryptographic service calls with referenced pages included in parenthesis. Section 2.2 addresses the database management functions in support of the cryptographic functions. Section 2.3 specifies the secret-key cryptographic service calls including message encryption, message integrity, and key management. Section 2.4 addresses the public-key cryptographic service calls including public-key encryption, digital signature, public key and certificate management, and public-key based key exchange service calls.

2 CRYPTOGRAPHIC SERVICE CALLS

The specification of the service calls is not intended to tie to any particular programming language. For each service call, the syntax of the call is presented first, followed by its parameter descriptions. Each parameter is listed with its data type and an indication of whether it is an input or output parameter, or both. It is possible for some input parameters to be passed through a trusted path such as a smart card other than from the application programs. For output parameter, whether it is a single-value parameter or an array of single-value elements, it is assumed that the host application program will allocate the necessary memory storage in advance to receive the output values.

Canonical representations are needed for certain data types such as cryptographic keys, algorithm parameters, and public key certificates. These data types are currently represented in the C language data structures as shown in the following. However, they may be converted to the ASN.1 representation if that is considered to be more appropriate. The data type of "string" refers to strings of characters or sequences of bytes. Strings are left justified, and padded on the right if necessary. Commands marked with an asterisk are restricted to cryptographic officers.

Data Structures for the Secret-Key Based Cryptographic Service Calls

```
#define    BYTE        unsigned char
#define    DES         0
#define    Skipjack    1
#define    KEK         0      /* Key encrypting key */
#define    KD          1      /* Data encryption key */
#define    DAC         2      /* DAC key */

typedef struct
{
        STRING      uid;
        STRING      keyid;
        BYTE        algid;  /* DES or Skipjack */
        BYTE        type;   /* KEK, KD or DAC key */
        BYTE        len;
        BYTE        parity;
        STRING      keyval;
        STRING      count;  /* key counter, if applicable */
} SecKeyRec;

typedef struct sknode
{
        SecKeyRec   current;
```

```
        struct sknode  *next;
} SecKeyList;
```

Data Structures for the Public-Key Based Cryptographic Service Calls

```
#define     BYTE            unsigned char
#define     UINT            unsigned int
#define     NAMESIZE        32
#define     MODULUS_SIZE    64
#define     DSA_Q_SIZE      20

typedef struct
{
        UINT  modulus_size;
        BYTE p[MODULUS_SIZE];
        BYTE q[DSA_Q_SIZE];
        BYTE g[MODULUS_SIZE];
        BYTE y[MODULUS_SIZE];
} DSAPublicKey;

typedef static struct
{
        BYTE x[DSA_Q_SIZE];
} DSAPrivateKey;

typedef struct
{
        UINT  modulus_size;
        BYTE n[MODULUS_SIZE];
        BYTE e[6];
        BYTE klen;   /* optional field */
} RSAPublicKey;

typedef static struct
{
        BYTE d[MODULUS_SIZE];
        BYTE p[MODULUS_SIZE/2];
        BYTE q[MODULUS_SIZE/2];
        BYTE d_p[MODULUS_SIZE/2]; /* D mod P-1 */
        BYTE d_q[MODULUS_SIZE/2]; /* D mod Q-1 */
        BYTE u[MODULUS_SIZE/2];  /* INVERSE of Q mod P */
```

```
        BYTE klen;
} RSAPrivateKey;    /* optional field */

typedef struct
{
        union {
                DSAPublicKey dsapubkey;
                RSAPublicKey rsapubkey;
        } pubktype;
} PublicKey;

typedef struct
{
        union {
                DSAPrivateKey dsaprikey;
                RSAPrivateKey rsaprikey;
        } priktype;
} PrivateKey;
```

/* The following defines the data type *Certificate* in the C language. Some of the supporting data types may not have been defined. */

```
typedef struct
{
        BYTE countryName[3];      /* country name */
        BYTE orgName[64];         /* organization name */
        BYTE orgUnitName[64];     /* optinal organization unit name */
        BYTE personalName[64];    /* personal name */
} Name;

typedef struct
{
        /* To be defined */
} UTCTime;

typedef struct
{
        UTCTime     notBefore;
        UTCTime     notAfter;
} Validity;

typedef struct
```

```
{
        Algorithm      algorithm;
        Parameters     parameters;
} AlgId;

typedef struct
{
        UINT                version;
        UINT                serialNumber;
        AlgId               signature;
        Name                issuer;
        Validity            validity;
        Name                subject;
        SubjPubKeyInfo      subjectPublicKeyInfo;
        UniqueId            issuerUniqueIdentifier;
        UniqueId            subjectUniqueIdentifier;
} Certificate;

typedef struct
{
        BYTE           uid[NAMESIZE];
        BYTE           keyid[NAMESIZE];
        BYTE           algid;
        BYTE           type;   /* key type or intended usage of key */
        PublicKey      pubkey;
        PrivateKey     prikey;
        Certificate    cert;
} PubKeyRec;

typedef struct pknode
{
        PubKeyRec      current;
        struct pknode  *next;
} PubKeyList;
```

2.1 List of Cryptographic Service Calls

Secret-Key Based Cryptographic Service Calls

Public-Key Based Cryptographic Service Calls

2.2 User Database Management Service Calls

VerifyUser

(

uid,	in/out	string
len,	input	integer
uauthent,	input	string
status	output	integer

)

Parameter description:

uid: specifies the address that points to the character string containing the user's identity.

len: specifies the length of *uauthent* in bytes.

uauthent:
specifies the address that point to the string of bytes containing the user's authenticator.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - User's identity is verified.
1 - Verification failed.
>1 - Abnormal termination

This service call verifies the authenticator (*uauthent*) of length *len* supplied by *uid* against the user's authenticator stored in the UDATABASE. A user's identity should be verified before any cryptographic request can be made. The result of the verification is returned in *status*.

***CreateUser**

(

uid,	in/out	string
utype,	input	character
len,	input	integer
uauthent,	in/out	string
status	output	integer

)

Parameter description:

uid: specifies the address that points to the character string containing the user's identity.

utype:
specifies the user type, for example, 'c' for cryptographic officers, 'u' for users.

len: specifies the length of uauthent in bytes.

uauthent:
specifies the address that points to the string of bytes containing the user's authenticator.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call creates an account for a cryptographic officer or a user under *uid* according to the user type indicated (*utype*). The CO's or the user's authentication information based on *uauthent* of length *len* is stored in the UDATABASE. It is recommended that *SetUserCommand* be called immediately after an account is created. The service call returns the resulting status to the calling program.

ChangeAuthent

(

oldlen,	input	integer
oldauthent,	input	string
newlen,	input	integer
newauthent,	in/out	string
status	output	integer

)

Parameter description:

oldlen:
specifies the length of oldauthent in bytes.

oldauthent:

specifies the address that points to the string of bytes containing the user's old authenticator.

newlen:
specifies the length of newauthent in bytes.

newauthent:
specifies the address that points to the string of bytes containing the user's new authenticator.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call lets a user change his/her authenticator. If the authenticator (*oldauthent*) of length *oldlen* supplied by the user is verified, the user's current authenticator is replaced by *newauthent* of length *newlen* and the resulting status is returned.

***SetUserCommand**

```
(
uid,        input       string
av,         input       string
status      output      integer
)
```

Parameter description:

uid: specifies the address that points to the character string containing the user's identity.

av: specifies the address that points to the string of bytes that is to receive the returned authorization vector. An authorization vector defines the service calls that a user can access. Each bit within the byte in the authorization vector corresponds to a service call, a one in the bit enables the corresponding service call whereas a zero disables it. For example, the correspondence between the service calls and their bit positions for the first byte of av looks as follows:

bit 0 - VerifyUser,
bit 1 - CreateUser,

bit 2 - ChangeAuthent,
bit 3 - SetUserCommand,
bit 4 - ShowUserCommand,
bit 5 - DeleteUser,
bit 6 - Logout,
bit 7 - Encipher.

It is assumed that a list of the service calls is available to the cryptographic officer.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call lets the CO set specific service calls that a user (*uid*) can access. The authorization vector (*av*) for user *uid* is stored in the UDATABASE, and the resulting status is returned.

ShowUserCommand

(

uid,	input	string
avlen,	input	integer
av,	output	string
status	output	integer

)

Parameter description:

uid: specifies the address that points to the character string containing the user's identity if the service call is executed by a co; null otherwise.

avlen:
specifies the total number of cryptographic service calls defined. Since each service call is represented by one bit in av as described in setusercommand, this parameter indicates how many bits of av to read which are meaningful.

av: specifies the address that points to the string of bytes containing the authorization vector associated with the user. "One" bits indicate enabled service calls whereas "zero" bits indicate disabled service calls.

status:

specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call uses *avlen* to determine how many bits of the authorization vector (*av*) of *uid* is to be read, and returns the authorization vector and resulting status to the calling program.

***DeleteUser**

(

uid,	input	string
status	output	integer

)

Parameter description:

uid:

specifies the address that points to the character string containing the name of the user whose record is to be removed from UDATABASE.

status:

specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call allows a CO to remove a user (*uid*) from UDATABASE. Every field in the database pertaining to the user is deleted and the storage is freed up. It should be noted that *DeleteKey* may need to be called before *DeleteUser* so that the user's keys are removed from SKEYDB before the user's account is closed. The resulting status is returned.

Logout

(

status	output	integer

)

Parameter description:

status:

specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call allows the user currently logged on to the CM to log out of the CM and returns the status to the calling program.

2.3 Secret-Key Based Cryptographic Service Calls

2.3.1 Encryption and Data Integrity Service Calls

Encipher

(

algid,	input	integer
mode,	input	integer
plen,	input	integer
pt,	input	string
keyid,	input	string
iv,	in/out	string
nbitfb,	input	integer
chain,	input	integer
clen,	output	integer
ct,	output	string
status	output	integer

)

Parameter description:

algid:

specifies the algorithm used for enciphering.
0 - DES [8]
1 - Skipjack [9]
>1 - reserved for future use

mode:

specifies the mode [10] of the enciphering operation.
0 - Electronic code book
1 - Cipher block chaining
2 - Cipher feedback

3 - Output feedback

plen:
specifies the length of the plaintext data in bytes.

pt: specifies the address that points to the string of bytes containing the plaintext data.

keyid:
specifies the address that points to the character string containing the name of the encrypting key.

iv: specifies the address that points to the string of bytes containing the 8-byte initialization vector. Used in mode 1, 2, and 3, null for mode 0.

nbitfb:
an integer between 1 and 64 indicating the number of bits of feedback to use in cipher feedback or output feedback mode. 0 in other cases.

chain:
specifies if chaining of consecutive encryption is desired. If chaining is desired, intermediate data values should be preserved across calls. This is useful for encrypting large files.

0 - Pt is the only block to be encrypted, i.e., it is the first and the last block.
1 - First block, but not the last.
2 - Middle blocks, i.e., not first, not last.
3 - Last block.

clen:
specifies the address that points to the data storage that is to receive the length of the resulting ciphertext in bytes.

ct: specifies the address that points to the string of bytes that is to receive the resulting ciphertext. Since ct is likely to contain nonprintable characters, it is necessary to use other routines to convert the string of packed bytes into a string of ascii hexadecimal characters when printing out the content of ct.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
1 - Output string (*ct*) size overflow

>1 - Other abnormal termination

This service call enciphers plaintext data (*pt*) of length *plen* in the specified algorithm (*algid*) and *mode* using *keyid* as the encryption key. For modes 2, 3, and 4, an initialization vector may be specified in the *iv* parameter. For cipher feedback, and output feedback modes, *nbitfb* specifies the number of bits of feedback to use. The ciphertext (*ct*), the length of the ciphertext (*clen*), and the *status* are returned to the calling program. Depending on the mode of operation, some padding may be added to the input plaintext data for a 64-bit block cipher, hence the length of the ciphertext (*clen*) may be greater than the length of the plaintext (*plen*). If status indicates a condition of string size overflow of the ciphertext (*ct*), the output parameter *clen* should indicate the length of the ciphertext and the calling program should increase the memory storage allocated for *ct* accordingly. When encrypting a large file, there may not be enough memory to hold the entire file, in this case, a means for chaining consecutive requests for multiple blocks is provided by the *chain* parameter. Depending on the value of this parameter, the CM would know when and when not to preserve intermediate values. If chaining is desired, the CM should preserve intermediate values. The distinction between the first block (chain set to 1) and the intermediate blocks (chain set to 2) can provide helpful information for the CM to implement the service call efficiently, since the first block usually requires initial setup which may not be needed for intermediate blocks.

Decipher

(

algid,	input	integer
mode,	input	integer
clen,	input	integer
ct,	input	string
keyid,	input	string
iv,	input	string
nbitfb,	input	integer
chain,	input	integer
plen,	output	integer
pt,	output	string
status	output	integer

)

Parameter description:

algid:

specifies the algorithm used for deciphering.
0 - DES

1 - Skipjack
>1 - reserved for future use

mode:
specifies the mode of the deciphering operation.
0 - Electronic code book
1 - Cipher block chaining
2 - Cipher feedback
3 - Output feedback

clen:
specifies the length of the ciphertext in bytes.

ct:
specifies the address that points to the string of bytes containing the ciphertext. Ct may contain nonprintable characters.

keyid:
specifies the address that points to the character string containing the name of the decrypting key.

iv: specifies the address that points to the string of bytes containing the 8-byte initialization vector. Used for mode 1, 2, and 3, null for mode 0.

nbitfb:
an integer between 1 and 64 indicating the number of bits of feedback to use for cipher feedback mode or output feedback mode. 0 for other cases.

chain:
specifies if chaining of consecutive decryptions is desired. If chaining is desired, intermediate data values should be preserved across calls. This is useful for decrypting large files.

0 - Ct is the only block to be decrypted, i.e., it is the first and the last block.
1 - First block, but not the last block.
2 - Middle blocks, i.e., not first, not last.
3 - Last block.

plen:
specifies the address that points to the data storage that is to receive the length of the plaintext in bytes.

pt: specifies the address that points to the string of bytes that is to receive the plaintext data.

status:
specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
1 - Output string (*pt*) size overflow
>1 - Abnormal termination

This service call decrypts the ciphertext (*ct*) of length *clen* in the specified algorithm (*algid*) and *mode* using *keyid* as the decrypting key. The input parameter *iv* specifies the initialization vector for modes 2, 3, and 4. For cipher feedback and output feedback modes, *nbitfb* specifies the number of bits of feedback to use. The decrypted plaintext (*pt*), the length of the plaintext (*plen*), and the resulting status are returned. The chaining parameter (*chain*) chains consecutive decryption requests for multiple blocks. Depending on the value of the parameter, the CM would know when and when not to preserve intermediate values across calls.

ComputeDAC

(

algid,	input	integer
len,	input	integer
data,	input	string
keyid,	input	string
chain,	input	integer
dac,	output	string
status	output	integer

)

Parameter description:

algid:
specifies the algorithm used for computedac.
0 - DES
>0 - reserved for future use

len: specifies the length of the data in bytes.

data:

specifies the address that points to the string of bytes containing the data for which DAC is to be computed.

keyid:

specifies the address that points to the character string containing the name of the key used for computing the DAC.

chain:

specifies if chaining of consecutive DAC operations is desired. If chaining is desired, intermediate data values should be preserved across calls.
0 - Data points to the only block whose DAC is to be computed.
1 - Data points to the first block, but not the last block.
2 - Data points to a middle block.
3 - Data points to the last block.

dac:

specifies the address that points to the string of packed bytes that is to receive the computed DAC. Since *dac* is likely to contain nonprintable characters, it is necessary to use other routines to convert the string of packed bytes into a string of ascii hexadecimal characters before the content of *dac* can be printed.

status:

specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call computes a Data Authentication Code (DAC) on *data* of length *len*, using the algorithm specified (*algid*) and *keyid* as the encrypting key. The computed *dac* and resulting status are returned to the calling program. Chaining consecutive DAC requests is provided by the *chain* parameter. If chaining is desired, the CM should preserve intermediate data values across consecutive calls.

VerifyDAC

(

algid,	input	integer
len,	input	integer
data,	input	string
keyid,	input	string
dac,	input	string
chain,	input	integer

status output integer
)

Parameter description:

algid:
specifies the algorithm used for verifydac.
0 - DES
>0 - reserved for future use

len: specifies the length of the data in bytes.

data:
specifies the address that points to the string of bytes containing the data whose DAC is to be verified.

keyid:
specifies the address that points to the character string containing the name of the key used for computing the DAC.

dac: specifies the address that points to the string of bytes containing the input data authentication code. If the user-entered data authentication code is a string of ascii hexadecimal characters with a blank space separating the left half and the right half of the code, it should be converted to a string of packed bytes first before calling verifydac.

chain:
specifies if chaining of consecutive calls is desired. If chaining is desired, intermediate data values should be preserved across calls.
0 - Data points to the only block whose DAC is to be verified.
1 - Data points to the first block, but not the last block.
2 - Data points to a middle block.
3 - Data points to the last block.

status:
specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - DAC is verified
1 - DAC is not verified
>1 - Abnormal termination

This service call first invokes *ComputeDAC* to compute the Data Authentication Code on *data* of indicated *len* using the algorithm specified (*algid*) and *keyid* as the encrypting key. The resulting DAC is checked to determine if it matches the input DAC (*dac*). If the two DACs are identical, verification succeeds. The result of the verification is returned in *status*. Chaining of consecutive *VerifyDAC* requests is provided by the chaining parameter (*chain*). If chaining is used, the CM should preserve intermediate data values across calls.

2.3.2 Key Management Service Calls

GenRandNum

```
(
seed,       input    string
len,        input    integer
randnum,    output   string
status      output   integer
)
```

Parameter description:

seed: specifies the seed in a string of bytes to reset the random-number generator. Null if random-number generator is not to be reset.

len: specifies the length of the random number in bits.

randnum:
specifies the address that points to the string of bytes that is to receive the generated random number.

status:
specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call generates a pseudo-random number *randnum* of specified length *len*. The random-number generator can be reset to a random starting point by using a new seed as specified in *seed*. The service call returns the generated random number and the resulting *status* to the calling program.

GenKey

```
(
keyid,        input    string
len,          input    integer
ktype,        input    integer
outputclear,  input    integer
ptkey,        output   string
status        output   integer
)
```

Parameter description:

keyid:
: specifies the address that points to the character string containing the name of the key to be generated.

len:
: specifies the length of key in bits. DES keys are 64 bits for a single key, 128 bits for a key pair.

ktype:
: specifies the type of key to be generated.
0 - key encrypting key
1 - data encryption key
2 - DAC key
3 - undetermined

outputclear:
: specifies if the generated plaintext key should be returned to the calling program. Regardless of whether the plaintext key is returned to the calling program, the newly generated key is always stored in SKEYDB.
0 - Do not return the plaintext key
1 - Return the plaintext key

ptkey:
: specifies the address that points to the string of bytes that is to receive the generated plaintext key if outputclear is set; null otherwise.

status:
: specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call generates a secret key of odd parity of length *len* and of type *ktype*. The generated key is stored under *keyid* in SKEYDB. If *outputclear* is set, the plaintext key (*ptkey*) and *status* are returned to the calling program; otherwise, *ptkey* contains a null pointer and only status is returned. The key may be initialized with a key counter using the *SetCount* service call.

DeleteKey

(

uid,	input	string
keyid,	input	string
status	output	integer

)

Parameter description:

uid: specifies the address that points to the character string containing the user's identity whose key is to be deleted from SKEYDB.

keyid:
specifies the address that points to the character string containing the name of the key to be deleted.

status:
specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
>0 - Abnormal Termination

This service call deletes the key identified in *keyid* of user *uid* from SKEYDB, and returns the resulting status to the calling program. A user can only delete his/her own keys, however, a CO may delete a user's keys before closing a user's account.

LoadKey

(

keyid,	input	string
len,	input	integer
ktype,	input	integer
key,	input	string
parity,	input	integer
status	output	integer

)

Parameter description:

keyid:
specifies the address that points to the character string containing the name of the key to be loaded.

len: specifies the length of key in bits. DES keys are 64 bits for a single key, 128 bits for a key pair.

ktype: specifies the type of key to be loaded.
0 - key encrypting key
1 - data encryption key
2 - DAC key
3 - undetermined

key: specifies the address that points to the string of bytes containing the clear key value to be loaded.

parity:
specifies if odd parity is to be used for the key to be loaded.
0 - Ignore parity checks
1 - set odd parity

status:
specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
>0 - Abnormal Termination

This service call loads a clear key (*key*) of length *len* and of type *ktype* to SKEYDB under the identity of *keyid*. If *parity* is set, the key is set to odd parity; otherwise, the parity of the key is not checked. The resulting status is returned to the calling program.

ShowKeyid

```
(
keys,       output      SecKeyList
status      output      integer
)
```

Parameter description:

keys: specifies the address that points to the data structure that is to receive the list of the logged-in user's keys currently loaded in the secure key storage.

status:
specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
>0 - Abnormal Termination

This service call allows a user to identify the secret keys that have been loaded in SKEYDB. Only the keys owned by the logged-in user should be identified and returned in the output data structure *keys*. At the minimum, the identifiers of the specific user's keys should be returned. Other keying information that does not reveal any sensitive information can also be returned. Plaintext key values should never be returned nor displayed. The resulting status is returned to the calling program.

ExportKey

```
(
keyid,      input     string
kkid,       input     string
nos,        input     integer
koffset,    input     integer
ori,        input     string
rcv,        input     string
len,        output    integer
enckey,     output    string
ktype,      output    integer
ctt,        output    string
status      output    integer
)
```

Parameter description:

keyid:
specifies the address that points to the character string containing the name of the key to be exported.

kkid:

specifies the address that points to the character string containing the name of the key encrypting key.

nos: specifies if notarization is desired or not.
0 - Notarization not desired
1 - notarization desired

koffset:
specifies if key offset is to be used.
0 - Key offset not desired
1 - key offset desired

ori: specifies the address that points to the character string containing the identity of the message originator.

rcv: specifies the address that points to the character string containing the identity of the message recipient.

len: specifies the address that points to the data storage that is to receive the length of the encrypted key (enckey) in bits. DES keys are 64 bits for a single-length key, 128 bits for a key pair.

enckey:
specifies the address that points to the string of bytes that is to receive the encrypted key value of *keyid*.

ktype: specifies the address that points to the data storage that will receive the key type of *keyid* that is to be exported.

ctt: specifies the address that points to a string of 7 bytes that is to receive the transmit count if nos or koffset is used; null otherwise.

status:
specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
>0 - Abnormal Termination

This service call encrypts a plaintext key before exporting the key outside the CM. If notarization (*nos*) is desired, a notarizing key is formed inside the CM before it is used to encrypt the key value of *keyid*. A notarizing key is formed by XORing the plaintext key value of *kkid* with a notary seal formed from the transmit count (*ctt*) of *kkid* and the identities of the message originator (*ori*) and the intended recipient (*rcv*). (See ANSI X9.17 Standard [5] for reference.) If key offset is desired, the plaintext value of *kkid* is XORed with the transmit count (*ctt*) of *kkid* before the result is used to encrypt the key value of *keyid* (see ANSI X9.17 Standard [5] for reference). The notarization flag (*nos*) and the key offset flag (*koffset*) are mutually exclusive, i.e. both flags can not be set to 1 in the same call. If neither key offset nor notarization is desired, the key value of *keyid* is simply encrypted by the key value of *kkid*, and the *ori* and *rcv* fields are ignored. The length of the encrypted key (*len*), the encrypted key (*enckey*), the type (*ktype*) of the exported key, and the resulting status are returned to the calling program. If notarization or key offset is used, *ctt* is also returned to the calling program; otherwise, a null pointer is returned.

ImportKey

(

keyid,	input	string
kkid,	input	string
len,	input	integer
enckey,	input	string
ktype,	input	integer
nos,	input	integer
koffset,	input	integer
ori,	input	string
rcv,	input	string
ctr,	input	string
parity,	output	integer
status	output	integer

)

Parameter description:

keyid:
specifies the address that points to the character string containing the name of the key to be imported.

kkid:
specifies the address that points to the character string containing the name of the key used to encrypt keyid.

len: specifies the length of the key to be imported in bits. DES keys are 64 bits for a single key, 128 bits for a key pair.

enckey:
specifies the address that points to the string of bytes containing the encrypted key value of *keyid*.

ktype: specifies the type of key that is to be imported.
0 - key encrypting key
1 - data encryption key
2 - DAC key
3 - undetermined

nos: specifies if notarization is to be used.
0 - No notarization used
1 - notarization used

koffset:
specifies if key offset is to be used.
0 - Key offset not desired.
1 - Key offset desired.

ori: specifies the address that points to the character string containing the identity of the message originator.

rcv: specifies the address that points to the character string containing the identity of the message recipient.

ctr: specifies the address that points to a string of 7 bytes containing the receive count used in key notarization.

parity:
specifies the address that points to the data storage that is to receive the parity conformance indicator of the imported key.
0 - Imported key not conformed to odd parity
1 - imported key conformed to odd parity

status:
specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
>0 - Abnormal Termination

This service call decrypts an imported key and stores it in SKEYDB. If notarization (*nos*) was used, a notarizing key was formed by XORing the key value of the key encrypting key (*kkid*) with a notary seal formed from the receive count (*ctr*) of *kkid* and the identities of the message originator (*ori*) and the recipient (*rcv*). The notarizing key is then used to decrypt the encrypted key (*enckey*). (For the processing of key counters and the notarization procedure, see the ANSI X9.17 Standard [5] for reference.) If key offset was used, the key value of *kkid* is XORed with the receive count (*ctr*) of *kkid* before the result is used to decrypt the *enckey*. The *ctr* is always compared with the stored receive count and processed according to the ANSI X9.17 standard. The notarization flag (*nos*) and the key offset flag (*koffset*) are mutually exclusive, i.e. both can not be set to 1 in the same call. If neither notarization nor key offset was used, the *ori, rcv*, and *count* fields are ignored and the *enckey* of length *len* is simply decrypted by *kkid*. The deciphered key is checked for odd parity and stored in SKEYDB under the identity of *keyid* and of type *ktype*. The result of the parity check and the *status* of processing the call are returned to the calling program.

XorKeys

```
(
newkeyid,     input      string
keyid1,       input      string
keyid2,       input      string
ktype,        input      integer
status        output     integer
)
```

parameter descriptions:

newkeyid:

specifies the address that points to the character string containing the name of the new key formed by xoring the keys of keyid1 and keyid2.

keyid1:

specifies the address of the character string containing the name of the key to be xored with the key of keyid2.

keyid2:

specifies the address of the character string containing the name of the key to be xored with the key of keyid1.

ktype: specifies the key type of *newkeyid* which is to be created by XORing two existing keys *keyid1* and *keyid2*.

0 - key encrypting key
1 - data encryption key

2 - DAC key
3 - undetermined

status:
specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
> 0 - Abnormal Termination

This service call exclusive-ors two keys, *keyid1* and *keyid2,* in SKEYDB to form a new key *newkeyid* and stores the new key with its type *ktype* in SKEYDB. The service call facilitates the use of dual control in forming a working key. The resulting status is returned to the calling program.

SetCount

(

kkid,	input	string
ctt,	input	string
ctr,	input	string
status	output	integer

)

parameter descriptions:

kkid:
specifies the address that points to the character string containing the name of the key encrypting key whose counters are to be reset.

ctt: specifies the address that points to a string of 7 bytes containing the transmit count to be set for kkid.

ctr: specifies the address that points to a string of 7 bytes containing the receive count to be set for kkid.

status:
specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call lets the calling program set the transmit count (*ctt*) and the receive count (*ctr*) associated with *kkid* in the SKEYDB. The resulting *status* is returned to the calling program. The *SetCount* and *ReadCount* service calls are added for compatibility with the ANSI X9.17 Key Management (Wholesale) Standard [5]. The initial values of key counters should be selected according to specifications in the X9.17 Standard. Processing of key counters in the specific cryptographic service calls must also follow the specifications in the X9.17 Standard.

ReadCount

```
(
kkid,       input       string
ctt,        output      string
ctr,        output      string
status      output      integer
)
```

parameter descriptions:

kkid:
specifies the address that points to the character string containing the name of the key encrypting key whose counters are to be retrieved.

ctt: specifies the address that points to a string of 7 bytes that is to receive the transmit count of kkid.

ctr: specifies the address that points to a string of 7 bytes that is to receive the receive count of kkid.

status:
specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call lets the calling program read the transmit count (*ctt*) and the receive count (*ctr*) associated with *kkid* in the SKEYDB. The counters and the resulting *status* are returned. *SetCount* and *ReadCount* service calls are added for compatibility with the ANSI X9.17 Key Management (Wholesale) Standard [5].

2.4 Public-Key Based Cryptographic Service Calls

2.4.1 Encryption and Digital Signature Service Calls

PubEncipher

(

algid,	input	integer
rcvpubkey,	input	PublicKey
plen,	input	integer
pt,	input	string
clen,	output	integer
ct,	output	string
status	output	integer

)

Parameter description:

algid: specifies the encryption algorithm to be used:
0 - RSA
>0 - reserved for future use

rcvpubkey:
specifies the address that points to the data structure containing the intended recipient's public key to be used for enciphering the message. Since only RSA is currently supported, the data type of this parameter is RSAPublicKey.

plen:
specifies the length of the plaintext data to be enciphered in bytes.

pt: specifies the address that points to the string of bytes containing the plaintext data. For the RSA encryption algorithm, the binary value of pt must be less than the binary value of the key modulus.

clen:
specifies the address that points to the data storage that is to receive the length of the resulting ciphertext in bytes.

ct: specifies the address that points to the string of bytes that is to receive the resulting ciphertext. For the RSA encryption algorithm, the binary value of ct would be less than the binary value of the key modulus.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.

0 - Service call executed correctly
>0 - abnormal termination

This service call uses the encryption algorithm specified in *algid* to encipher a message one block at a time. Currently only the RSA algorithm is supported. The public key of the intended recipient *rcvpubkey* is used to encipher the plaintext message *pt* of length *plen*. The resulting ciphertext *ct* of length *clen* and *status* are returned.

PubDecipher

(

algid,	input	integer
rcvid,	input	string
rcvprikid,	input	string
clen,	input	integer
ct,	input	string
plen,	output	integer
pt,	output	string
status	output	integer

)

Parameter description:

algid: specifies the decryption algorithm to be used:
0 - RSA
>0 - reserved for future use

rcvid:
specifies the address that points to the character string containing the message recipient's identity. This parameter and the parameter *rcvprikid* are used by the Cryptographic Module to retrieve the message recipient's private key from the secure key storage in order to decipher the received ciphertext.

rcvprikid:
specifies the address that points to the character string containing the identity of the message recipient's private key. The Cryptographic Module is responsible for retrieving the private key from the secure key storage before decryption can take place.

clen: specifies the length of the ciphertext in bytes.

ct: specifies the address that points to the string of bytes containing the ciphertext. For the RSA algorithm, the binary value of *ct* should be less than the binary value of the key modulus.

plen:
specifies the address that points to the data storage that is to receive the length of the resulting plaintext in bytes.

pt: specifies the address that points to the string of bytes that is to receive the resulting plaintext. For the RSA algorithm, the binary value of pt would be less than the binary value of the key modulus.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - abnormal termination

This service call uses the algorithm specified in *algid* to decipher a message one block at a time. Currently only the RSA algorithm is supported. The ciphertext *ct* is decrypted using the message recipient's private key identified in *rcvprikid*. The CM is responsible for retrieving the private key from the secure key storage. If not readily available, the private key can be retrieved by using the message recipient's identifier *rcvid* and the private key identifier *rcvprikid* as indices to the secure key storage. To decipher a message using the RSA algorithm, the message recipient's public key is also needed. It is assumed that the message recipient's public and private key pair is stored side by side as a single record, thus retrievable as a single record as implied in the data structure *PubKeyRec* described earlier. The resulting plaintext *pt*, its length *plen*, and status are returned.

Hash

(

algid,	input	integer
msglen,	input	integer
msg,	input	string
mdlen,	output	integer
md,	output	string
status	output	integer

)

Parameter description:

algid: specifies the hashing algorithm to be used:
0 - SHA [11]
1 - MD2
2 - MD5
>2 - reserved for future use

msglen:
specifies the length of the message to be hashed in bytes.

msg: specifies the address that points to the string of bytes containing the message to be hashed.

mdlen:
specifies the address that points to the data storage that is to receive the length of the computed message digest in bytes.

md: specifies the address that points to the data storage that is to receive the computed message digest.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - abnormal termination

This service call uses the algorithm specified in *algid* to hash a message *msg* of length *msglen* to a message digest *md*. A message longer than the modulus size must be hashed before it is signed. The length of the message digest *mdlen*, the message digest *md*, and the resulting *status* are returned.

PreSign

(

algid,	input	integer
count,	input	integer
uid,	input	string
prikeyid,	input	string
status	output	integer

)

Parameter description:

algid:
specifies the algorithm for signature precomputation, currently only DSS [12] can take advantage of precomputation:
0 - DSS
>0 - reserved for future use

count:
specifies the number of precomputations desired.

uid: specifies the address that points to the character string containing the user's identity whose private key will be used with the precomputed value in order to compute a digital signature. If the public key components P, Q, and G are to be shared globally, then this parameter points to a null string.

prikeyid:
specifies the address that points to the character string containing the identity of the private key which is to use the precomputed value for signature generation. If the public key components P, Q, and G are to be shared globally, this parameter points to a null string.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - abnormal termination

This service call uses the algorithm specified in *algid* to perform signature precomputation. Currently only the DSS algorithm can take advantage of signature precomputation. The number of precomputations desired is indicated in *count*. It is assumed that the precomputed values will be stored at specific places where the Cryptographic Module can retrieve these values automatically if such values exist. If the public key parameters p, g, and q are not shared globally, these parameters must be stored with the individual user's public key. In this case, the CM uses the information supplied in *uid* and *prikeyid* to determine the specific key the precomputation is intended for. The CM is responsible for maintaining the link between precomputed values and the specific key the precomputation was performed for. Precomputed values are used in a First In First Out fashion until exhausted. The resulting status is returned.

***SetPubParam**

(
algid, input integer

pubparam, input PublicKey
status output integer
)

parameter descriptions:

algid: specifies the public key algorithm to be used:
0 - DSA
>0 - reserved for future use

pubparam:
specific to the DSA algorithm, this parameter specifies the address that points t o the public key data structure containing the public key parameters *p, q, g* to be set.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - abnormal termination

This service call sets the public key parameters that can be shared by a group of users. Currently only the DSA algorithm has global parameters *p, q,* and *g* that need to be s et, therefore, *algid* should indicate DSA. The global parameters are stored by the CM at designated places. Since the calling program need not access these parameters directly, no handle is returned to the calling program. This service call should be invoked once for each group of users who are to share the parameters, and the service call should be made before the first public key pair is generated (by calling *GenPubKey*) for the group. These parameters, once set, can not be changed unless new keys are generated for the entire group of users. The service call returns the resulting status to the calling program.

ReadPubParam
(
algid, input integer
pubparam, output PublicKey
status output integer
)

parameter descriptions:

algid: specifies the public key algorithm to be used:

0 - DSA
>0 - reserved for future use

pubparam:
specific to the DSA algorithm, this parameter specifies the address that points to the data structure that is to receive the DSA public key parameters *p, q, g*.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - abnormal termination

This service call reads the global parameters associated with the algorithm specified in *algid*. Currently this service call only applies to the DSA algorithm, and it returns the parameters *p, q, g* in a data structure *pubparam* and the resulting status to the calling program.

Sign

(

algid,	input	integer
mdlen,	input	integer
md,	input	string
prikeyid,	input	string
siglen,	output	integer
signature,	output	string
status	output	integer

)

parameter descriptions:

algid: specifies the algorithm to be used for signing:
0 - DSA
1 - RSA
>1 - reserved for future use

mdlen:
If a message is hashed before it is signed, *mdlen* specifies the length of the message digest in bytes; otherwise it specifies the length of the message in bytes.

md: specifies the address that points to the string of bytes containing the message or message digest to be signed. If hashing of the message is not needed, the entire message is signed; otherwise, the message digest is signed.

prikeyid:
specifies the address that points to the character string containing the identity of the signer's private key to be used for signing. It is assumed that the CM will use the currently logged-in user's identifier and the specified private key identifier to retrieve the particular key for signing.

siglen:
specifies the address that points to the data storage that is to receive the length of the resulting signature.

signature:
specifies the address that points to the string of bytes that is to receive the resulting digital signature.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - abnormal termination

This service call signs a message or a message digest *md* of length *mdlen* using the algorithm indicated in *algid*. If a message is longer than the modulus size, the message must be hashed before it is signed. The CM uses the currently logged-in user's id and the private key id identified in *prikeyid* to retrieve the particular key for signing. The service call returns the resulting signature length *siglen*, the signature, and status to the calling program. Note that the binary value of the message digest must be less than or equal to the binary value of the key modulus.

VerifySig
(

algid,	input	integer
mdlen,	input	integer
md,	input	string
pubkey,	input	PublicKey
siglen,	input	integer
signature,	input	string
status	output	integer

)

Parameter description:

algid:
specifies the algorithm used for signature verification:
0 - DSA
1 - RSA
>1 - reserved for future use

mdlen:
If the message was hashed before it was signed, this parameter specifies the length of the message digest in bytes; otherwise, it specifies the length of the message in bytes.

md: Depending on whether a message was hashed or not, this parameter specifies the address that points to the string of bytes containing the message or message digest that was signed.

pubkey:
specifies the address that points to the data structure containing the signer's public key.

siglen:
specifies the length of the signature to be verified in bytes.

signature:
specifies the address that points to the string of bytes containing the digital signature to be verified.

status:
specifies the address that points to the data storage that is to receive the result of the service call.
0 - Signature verified
1 - Verification failed
>1 - abnormal termination

Using the algorithm specified in *algid*, this service call verifies the signature (of length *siglen*) generated on a message/message digest *md* (of length *mdlen*) using the signer's public key (*pubkey*). The binary value of the message digest must be less than or equal to the binary value of the key modulus. The result of the verification is returned in *status*.

2.4.2 Key management service calls

GenPubKey

(

algid,	input	integer
uid,	input	string
modulus,	input	integer
pubkeyid,	input	string
ktype,	input	integer
pubkey,	output	PublicKey
status	output	integer

)

Parameter description:

algid:

specifies the algorithm for which the generated public key pair will be used.
0 - DSA
1 - RSA
>1 - reserved for future use.

uid:

specifies the address that points to the character string containing the user's identity for whom the key pair is generated. If the security policy in force allows a security officer to generate a public key pair for an end user, this parameter specifies the end user's identity; otherwise, it specifies the identity of the currently logged-in user.

modulus:

specifies the size of the key modulus in bytes.

pubkeyid:

specifies the address that points to the character string containing the name to be used for the generated public key pair. It is assumed that the generated key pair will share the same name.

ktype: specifies the intended usage of the public key pair that is to be generated.
0 - public-key based encryption key
1 - signature key
2 - key exchange
3 - undetermined

pubkey:

specifies the address that points to the data structure that is to receive the generated public key pair.

status:

specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - abnormal termination

Based on the algorithm (*algid*) and *modulus* specified, this service call generates a public-private key pair for the user specified in *uid*. If the security policy in force allows a security officer to generate a public key pair for an end user, the parameter *uid* specifies the identity of the end user for whom the key pair is generated. Otherwise, it specifies the identity of the currently logged-in user. The generated key pair is stored in secure storage under the name specified in *pubkeyid* along with the intended usage of the key as indicated in *ktype*. The service call returns the public key data structure (*pubkey*) and the resulting status to the calling program.

LoadPubKey

```
(
pubkey,     input      PubKeyRec
status      output     integer
)
```

Parameter description:

pubkey:

specifies the address that points to the data structure containing the public key and associated keying information which is to be loaded into the secure key storage.

status:

specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - abnormal termination

This service call loads the public key and associated keying information of a particular user into the public key storage and returns the resulting status to the calling program.

ShowPubKey

(
pubkeys, output PubKeyList
status output integer
)

Parameter description:

pubkeys:

specifies the address that points to the data structure that is to receive the list of public keys that are currently loaded in the public key data storage.

status:

specifies the address that points to the data storage that is to receive the status of processing the service call.
0 - Service call executed correctly
>0 - Abnormal Termination

This service call allows a user to see the public keys that have been loaded in the public key storage that the user can access to. At a minimum, the public key identifier and the public key value of each loaded public key should be returned in the output data structure *pubkeys*. Other keying information that does not reveal any sensitive information can also be returned. The plaintext private key(s) of the user should never be returned. The service call returns the resulting status to the calling program.

RetrvPubKey

(
uid, input string
algid, input integer
pubkeyid, input string
pubkey, output PublicKey
status output integer
)

Parameter description:

uid: specifies the address that points to the character string containing the user's identity whose public key is to be retrieved.

algid:

specifies the algorithm that the retrieved public key is to be used for.

pubkeyid:
specifies the address that points to the character string containing the identity of the user's public key to be retrieved.

pubkey:
specifies the address that points to the public key data structure that is to receive the retrieved public key.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - abnormal termination

This service call retrieves the public key (*pubkey*) of user *uid* and of the specified algorithm (*algid*) from the public key storage. The specific public key to be retrieved is identified in *pubkeyid*. The service call returns the retrieved public key and the resulting status to the calling program.

DeletePubKey

```
(
uid,          input      string
algid,        input      integer
pubkeyid,     input      string
status        output     integer
)
```

Parameter description:

uid: specifies the address that points to the character string containing the user's identity whose public key pair is to be deleted from the public key storage.

algid:
specifies the algorithm that the public key to be deleted is associated with.

pubkeyid:
specifies the address that points to the character string containing the identity of the public key pair to be deleted.

status:

specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - abnormal termination

This service call deletes the public-private key pair and the associated public key certificate of user *uid* identified in *pubkeyid* and associated with algorithm *algid* from the public key storage. It is assumed that a certificate is stored with its associated public key. Therefore, this service call deletes the entire key pair and the associated keying information including the certificate from the key storage. The service call returns the resulting status to the calling program.

LoadCert

(

uid,	input	string
algid,	input	integer
pubkeyid,	input	string
cert,	input	Certificate
status	output	integer

)

Parameter description:

uid: specifies the address that points to the character string containing the user's identity whose public key certificate is to be stored.

algid:
specifies the algorithm that the public key certificate is associated with.

pubkeyid:
specifies the address that points to the character string containing the identity of the public key whose certificate is to be stored.

cert:
specifies the address that points to the data structure containing the user's public key certificate that is to be stored.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly

>0 - abnormal termination

This service call allows a user to store his own and other users' public key certificates in the public key storage. The service call loads the public key certificate *cert* of user *uid* associated with the algorithm *algid* in the key storage. It is assumed that a certificate is stored side by side with the associated public key identified in *pubkeyid*. Thus, a certificate can be referenced by the user ID and the public key ID. The service call returns the resulting status to the calling program.

RetrvCert

(

uid,	input	string
algid,	input	integer
pubkeyid,	input	string
cert,	output	Certificate
status	output	integer

)

Parameter description:

uid: specifies the address that points to the character string containing the user's identity whose public key certificate is to be retrieved.

algid:
specifies the algorithm that the public key certificate is associated with.

pubkeyid:
specifies the address that points to the character string containing the identity of the public key whose certificate is to be retrieved.

cert:
specifies the address that points to the data structure that is to receive the retrieved certificate.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - abnormal termination

This service call retrieves the public key certificate *cert* of user *uid* associated with the algorithm *algid* from the key storage. It is assumed that a certificate is stored side by side with its associated public key which is identified in *pubkeyid*. Therefore, a certificate can be retrieved just like a public key by specifying the user ID and the public key ID. The service call returns the retrieved certificate and resulting status to the calling program.

PubExportKey

(

algid,	input	integer
rcvpubkey,	input	PublicKey
exkeytype,	input	integer
exkeyid,	input	string
exkeylen,	output	integer
exkey,	output	string
status	output	integer

)

Parameter description:

algid:

specifies the algorithm to be used for key exchange:
0 - Diffie Hellman key exchange algorithm
1 - RSA
>1 - reserved for future use

rcvpubkey:

specifies the address that points to the data structure containing the public key of the entity that is to receive the secret key component and import the secret key. The receiver's public key is used in computing the secret key component that is to be sent from the key exporter (sender) to the key importer (receiver).

exkeytype:

specifies the type of secret key to be exported.
0 - DES key
1 - Skipjack key
>1 - Reserved for future use

exkeyid:

specifies the address that points to the character string containing the identity to be used for the exchanged secret key.

exkeylen:

specifies the address that points to the data storage that is to receive the length of the secret key component to be exported in bytes. This length should be no more than the size of the key modulus.

exkey:

specifies the address that points to the string of bytes that is to receive the secret key component which will be sent to a key sharing party. The key sharing party is referred to as the key importer or the receiver in this document. For the Diffie Hellman algorithm, both the sending party (key exporter) and the receiving party must derive the secret key to be shared between them. The receiver shall use the quantity received (*exkey*) and his/her private key to derive the secret key, whereas the sender uses the receiver's public key to derive the secret key (See the functional description of this service call for details). Therefore, for the Diffie Hellman algorithm, this quantity is NOT the encrypted value of the secret key that is to be established.

If *algid* specifies the RSA algorithm, then the secret key is assumed to have been generated and is identified in *exkeyid*. In this case, the quantity *exkey* is the encrypted key value of the secret key to be shared. The RSA algorithm is used in encrypting the secret key.

status:

specifies the address that points to the data storage that is to receive the result of processing the service call.
0 - Service call executed correctly
>0 - abnormal termination

This service call provides a capability for applications to exchange a secret key using a public-key based algorithm. Currently two algorithms are supported [6].

If *algid* specifies the Diffie Hellman key exchange algorithm, then *exkey* is the quantity c computed by $c = g^k \bmod p$, where g and p are the importer's public key components as specified in *rcvpubkey* and k is a one-time random number generated by the key exporter for each key exchange.

The quantity *exkey* is sent, or exported, to another party to derive the secret key to be shared with the sending party. Both the sender and recipient must derive the secret key. The sender, or the key exporter, derives the secret key K by computing $K = y^k \bmod p$. The key importer derives the secret key K by computing $K = c^x \bmod p$, using the received quantity c (which is *exkey*) and his/her private key. It is assumed that this service call will derive the secret key automatically for the key exporter, and store the secret key in a secure storage using the name specified in *exkeyid*. It is further assumed that the CM has access to both the secret key and the public key databases.

It should be noted that the length of the computed K may be longer than what is needed for a secret key, therefore, only a portion of the computed K is used. The type of secret key to be exchanged is indicated in *exkeytype*. If a DES key is to be exchanged, the least significant 56 bits of the computed K is used. If a Skipjack key is desired, the least significant 80 bits is used. The length of *exkey* should be no more than the size of the importer's key modulus. This length is returned in *exkeylen* along with *exkey*, the status of the call is also returned to the application program.

If *algid* specifies the RSA algorithm, the representations of the parameters are quite different. In this case, *exkeyid* identifies the secret key that has been generated and is to be retrieved from the secure secret key storage using the key exporter's ID and the secret key ID specified in *exkeyid*. The retrieved secret key is encrypted under the importer's public key specified in *rcvpubkey*, and the resulting *exkey* is sent to the key importer. The type of secret key that is to be exchanged is indicated in *exkeytype*. The length of the RSA-encrypted key (*exkeylen*) should be no more than the modulus size of the importer's public key. The service call returns the length of the RSA-encrypted key, the encrypted key *exkey*, and the resulting status to the application program.

Though it is beyond the scope of this service call, the application program should send the following information to the key importer: algid, exkeytype, exkeyid, importer's public key id (so that the importer knows which public and private key to use to derive the secret key), exkeylen, and exkey.

PubImportKey

```
(
algid,          input     integer
impkeytype,     input     integer
impkeyid,       input     string
pubkeyid,       input     string
impkeylen,      input     integer
impkey,         input     string
status          output    integer
)
```

Parameter description:

algid:

specifies the algorithm used for key exchange:

0 - Diffie Hellman key exchange algorithm
1 - RSA
>1 - reserved for future use

impkeytype:
specifies the type of secret key to be imported.
0 - DES key
1 - Skipjack key
>1 - Reserved for future use

impkeyid:
specifies the address that points to the character string containing the identity to be used for the imported key.

pubkeyid:
specifies the address that points to the character string containing the identity of the public key which was used in computing the quantity *exkey* in *PubExportKey*. (This identifier should have been sent from the key exporting application and received by the key importing application before the importing application can make a call to *PubImportKey*.) The same public key used in computing *exkey* and its associated private key shall be used in deriving the secret key to be shared. The parameter is needed because an entity may have multiple public keys.

impkeylen:
specifies the length of secret key component (*impkey*) received from the key exporter in bytes. This length should be no more than the modulus size of the public key identified in *pubkeyid*.

impkey:
specifies the address that points to the string of bytes containing the secret key component that is received from the key exporter.

For the Diffie Hellman algorithm, this quantity is used along with the importer's private key (associated with the public key identified in *pubkeyid*) to derive the secret key.

For the RSA algorithm, this quantity is the RSA-encrypted key value of the secret key that is to be shared with the key exporter.

status:
specifies the address that points to the data storage that is to receive the result of processing the service call.

0 - Service call executed correctly
>0 - abnormal termination

This service call allows an application program to import a secret key after receiving encrypted keying information (referred to as the *secret key component* in *PubExportKey* and *PubImportKey*) from a key exporter. All the input parameters listed for the call should come from the key exporter so that the exchanged key can be imported properly. Currently two algorithms are supported for key exchange. In either case, *algid* specifies the particular key exchange algorithm to be used in importing the key, and the type of key to be imported is indicated in *impkeytype*. The key importer uses the information specified in *pubkeyid* to find the associated private key in order to derive the secret key. The derived secret key is then stored under the identity specified in *impkeyid*. Depending on the algorithm chosen, the parameter *impkey* represents different information.

If *algid* specifies the Diffie Hellman key exchange algorithm, then the received *impkey* of length *impkeylen* is the quantity c computed by $c = g^k \bmod p$, where g and p are the public key components of the key importer and k is a one-time random number generated by the key exporter for each key exchange. The key importer derives the secret key K by computing $K = c^x \bmod p$, using the received *impkey* and the key importer's private key *x*.

It should be noted that the length of the computed K may be longer than what is needed for a secret key, therefore, only a portion of the computed K is used. If a DES key is imported, the least significant 56 bits of the computed K is used. If a Skipjack key is desired, the least significant 80 bits is used. Once the secret key is imported and stored in the secret key storage, the service call returns the resulting status to the application program.

If *algid* specifies the RSA algorithm, the received quantity *impkey* is the RSA-encrypted key value of the secret key which is to be imported. The key importer should use its private key associated with the public key identified in *pubkeyid* to RSA-decrypt the quantity *impkey* in order to derive the secret key. As stated earlier, only a portion of the decrypted key value is needed for the secret key depending on the type of secret key that is exchanged. Once the secret key is imported and stored in the secret key storage, the service call returns the resulting status to the application program.

3 REFERENCES

[1] FIPS PUB 113, Computer Data Authentication, National Bureau of Standards, May 1985.

[2] FIPS PUB 140-1, Security Requirements for Cryptographic Modules, National Institute of Standards and Technology, January 1994.

[3] Deborah Russell and G. T. Gangemi Sr., Computer Security Basics, O'Reilly & Associates, Inc., 1991.

[4] P. Fahn, Answers to Frequently Asked Questions about Today's Cryptography, RSA Laboratories, September 1992.

[5] ANSI X9.17 Financial Institution Key Management (Wholesale) Standard, American Bankers Association, Approved April 1985, Reaffirmed 1991.

[6] NIST Special Publication 800-2, Public-Key Cryptography, National Institute of Standards and Technology, April 1991.

[7] James Nechvatal, A Public-Key Certificate Management System, National Institute of Standards and Technology, May 1992.

[8] FIPS PUB 46-2, Data Encryption Standard, National Institute of Standards and Technology, Approved January 1977, Reaffirmed December 1993.

[9] FIPS PUB 185, Escrowed Encryption Standard, National Institute of Standards and Technology, February 1994.

[10] FIPS PUB 81, DES Modes of Operation, National Bureau of Standards, November 1981.

[11] FIPS PUB 180, Secure Hash Standard, National Institute of Standards and Technology, May 1993.

[12] FIPS PUB XXX, Draft Digital Signature Standard, National Institute of Standards and Technology, August 1991.