CGX Cryptographic Module Security Policy

version 1.24

last revision 24 March 2000

Information Resource Engineering, Inc.

This document may be reproduced only in its entirety, without revision.

Document Author: Jeremy Lapon

Information furnished in this document is preliminary. No responsibility is assumed by Information Resource Engineering for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Information Resource Engineering, Inc

Table of Contents

A. Scope of Document 3
APPLICATION LAYER 3
CGX COMMAND INTERFACE LAYER 3
CGX COMMAND PROCESSOR LAYER 4
CGX OVERLAY LAYER 4
CRYPTOLIB LAYER 5
B. Security Level 6
C. Roles and Services 7
Command Descriptions: 9
D. Security Rules 13
E. Definition of Security Relevant Data Items 15
SYMMETRIC KEYS 15
G. Service to SRDI Access Operation 16
H. List of Acronyms 17

A. Scope of Document

The IRE security software which is made available to applications running on a Windows 95, 98 or NT 4.0 platform, as well as, the ADSP 2141 Safenet/DSP cryptographic processor, is designated the SafeNet CGX (**C**rypto**G**raphic e**X**tensions) Kernel. (Running on the ADSP 2141 Safenet/DSP cryptographic processor is not included as part of this FIPS 140-1 validation of the CGX software.) It is a suite of approximately 40 functions, which are available to applications which require security services. To simplify application-level access to crypto functions an Application Programming Interface (API) is provided to the CGX Kernel. The CGX Command Interface defines the boundaries between the security functions (which the CGX Kernel implements) and the externally running applications. One of the primary goals of the CGX software is to abstract the CGX Kernel from the application in a secure and efficient manner. The CGX interface is designed so that it can be viewed as a Crypto Library with a C-structure like interface with argument and pointer-passing. To make a CGX command call, a structure is populated with arguments and a call is made to the CGX kernel, passing a pointer to the structure.

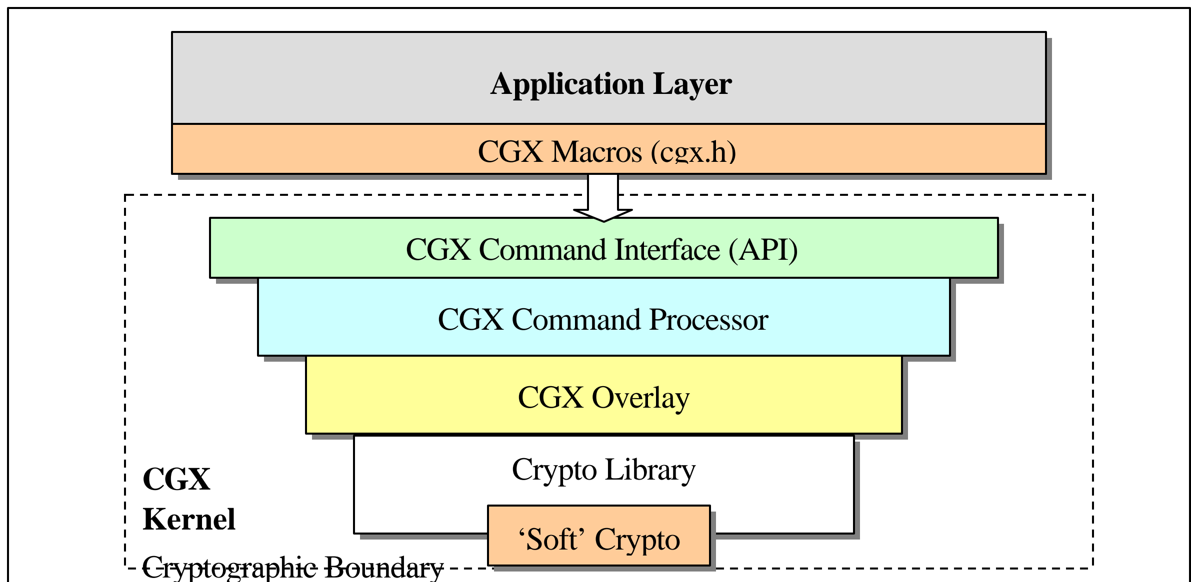


Figure 1 CGX Software Layers

The CGX software resides within the dashed line illustrated in Figure 1. The application uses the CGX Command Interface as an API to access the CGX command set. To better understand the software architecture of the CGX security software, a description of each layer is provided in the sub-sections below.

APPLICATION LAYER

The application layer is where the actual application program and data space resides. In order to access the cryptographic services, the application must invoke the command interface and pass-in a command code and arguments.

Residing, as part of the application layer are the macro functions, which IRE provides in its cgx.h file. These optional macros assist the application in preparing the command messages prior to calling the CGX kernel.

CGX COMMAND INTERFACE LAYER

The CGX command interface layer is an Application Programming Interface (API) which defines the boundaries between the application and the CGX Kernel. The CGX command interface provides the mechanism to enter and exit the CGX Kernel to execute a specific cryptographic command.

The software interface to the CGX Kernel is via a pair of data structures called the *kernel block* and the *command block*. The *kernel block* is a simple structure that specifies memory modes and provides a pointer to the *command block*, allowing flexible placement in memory. It also contains a status element, which the application can read to determine the result status of a requested cryptographic service. The *command block* is used to request a specific cryptographic command and to provide a means to pass-in arguments.
Therefore, all communications between the application and CGX Kernel is via the command interface and a *Kernel block* and *command block*.

CGX COMMAND PROCESSOR LAYER

The CGX Command Processor implements a secure Operating System responsible for processing application requests for various cryptographic services. Once the CGX Kernel is active, it can process the requested cryptographic function specified in the *kernel block* & *command block* defined as part of the CGX command interface layer. The CGX Command Processor is responsible for maintaining the security of the internal cryptographic software, key material, and associated security devices.

Like other operating systems, the CGX Command Processor is responsible for time-sharing the security resources. It does this through preemption management and system integrity management.

Preemption Management: For certain CGX commands, the Command Processor can allow a new command to preempt a running one. This feature is provided because it may be desirable to nest a 2nd CGX command on top of one already running. However, if a preemptive request comes in and the CGX Kernel is not executing one of the preemptable commands, the CGX Command Processor will return a `CGX_BUSY_S` status code and will not execute the command.

System Integrity Management: Lastly, the CGX Command Processor is responsible for monitoring the security integrity of CGX. As part of initialization processing, the CGX Command Processor runs a suite of self-tests to verify the health of the security components. The CGX Command Processor will not give control of the CGX Kernel to the application until the self-test suite completes successfully. This level of control is only permitted for the CGX Command Processor; thus preventing accidental or intentional access to areas of the security blocks not allowed.

CGX OVERLAY LAYER

The CGX overlay layer is provided as the interface into IRE's CryptoLIB software. The CryptoLIB software is a library that is designed for multiple platforms, ranging from the PC to embedded systems. The CGX overlay acts as the 'wrap code' to enable the library to execute on any platform unmodified.

Figure 2 illustrates the data flow through the CGX overlay layer.

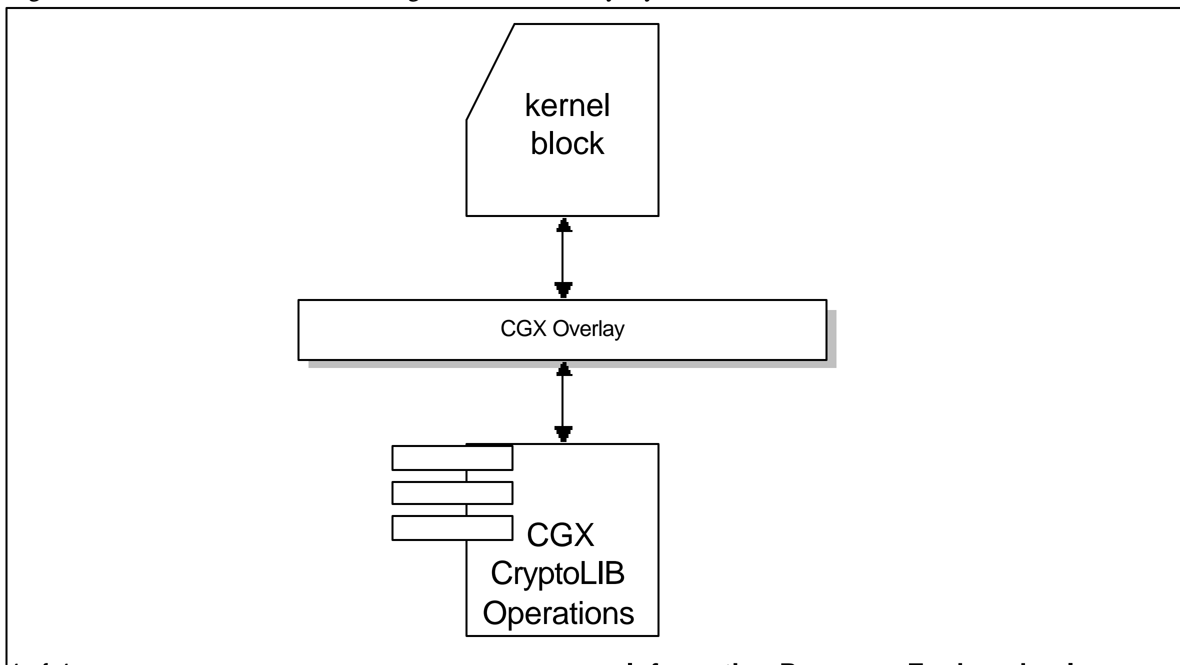


Figure 2

When a cryptographic request is received, the CGX Command Processor parses the *kernel block* to determine the cryptographic command to execute. The CGX Command Processor executes a CGX overlay operation from a table, based on the command value embedded in the *command block* portion of the *kernel block*. The CGX overlay operation is responsible for extracting the arguments from the *kernel block* and invoking the proper CryptoLIB operations. In some cases, the CGX overlay operation may invoke several CryptoLIB operations. In effect, this is an object-oriented approach where the CGX overlay class is the parent class to the CryptoLIB classes.

CRYPTOLIB LAYER

The CryptoLIB layer contains IRE's Crypto Library software as well as 'soft' versions of the hardware algorithms (the soft versions are ifdef'ed out at compile time for the ADSP 2141 hardware cryptographic processor). The CryptoLIB software is a library of many cryptographic classes implementing various cryptographic algorithms from symmetrical encryption algorithms to one-way Hash functions, to public key operations.

The CryptoLIB API is transparent to whether it is running with hardware acceleration or utilizing the library's software crypto functions. This hides the implementation of the ADSP 2141 platform and allows full reuse of the general CryptoLIB software. This provides several advantages including:

- The CryptoLIB software can be independently used in the PC environment
- The software only 'C' version of CryptoLIB can be used during development of products, which will use the ADSP 2141. Enhancements and modifications to the CryptoLIB can be made and tested in the PC environment before migrating them into the hardware of the ADSP 2141

B. Security Level

The cryptographic module meets the overall requirements applicable to Level 2 security of FIPS 140-1 when running on Windows NT 4.0. The module meets Level 1 when running on Windows 95/98 or Windows NT configured for single user mode.

Table 1. Module Security Level 2 Specification

Security Requirements Section	Level
Cryptographic Module	2
Module Interfaces	2
Roles and Services	2
Finite State Machine	2
Physical Security	2
Software Security	3
Operating System Security	2
Key Management	2
Cryptographic Algorithms	2
EMI/EMC	3
Self Test	4

Table 2. Module Security Level1 Specification

Security Requirements Section	Level
Cryptographic Module	1
Module Interfaces	1
Roles and Services	1
Finite State Machine	1
Physical Security	1
Software Security	3
Operating System Security	1
Key Management	1
Cryptographic Algorithms	1
EMI/EMC	3
Self Test	4

C. Roles and Services

The CGX cryptographic module shall support two distinct operator roles. These operator roles are:

1. User Role
2. Cryptographic Officer Role

The CGX library running on Windows 95/98 does not incorporate either **role-based** authentication or **identity-based** authentication. This is targeted for Level 1 security.

When running on Windows NT, the cryptographic module enforces **role-based** operator authentication. Windows NT provides authenticated login, which it enforces on the operator.

The following table lists the CGX commands and their applicable roles.

CGX Command	Crypto Officer Role	User Role
CGX_INIT	X	X
CGX_DEFAULT	X	X
CGX_RANDOM	X	X
CGX_GET_CHIPINFO	X	X
CGX_ZEROIZE_KEYS	X	X
CGX_SELF_TEST	X	X
Symmetric Key Commands		
CGX_UNCOVER_KEY	X	X
CGX_GEN_KEK	X	X
CGX_GEN_KEY	X	X
CGX_LOAD_KEY	X	X
CGX_DERIVE_KEY	X	X
CGX_TRANSFORM_KEY	X	X
CGX_EXPORT_KEY	X	X
CGX_IMPORT_KEY	X	X
CGX_DESTROY_KEY	X	X
CGX_LOAD_KG	X	X
CGX_ENCRYPT	X	X
CGX_DECRYPT	X	X
Asymmetric Key Commands		
CGX_GEN_PUBKEY	X	X
CGX_GEN_NEWPUBKEY	X	X
CGX_GEN_NEGKEY	X	X
CGX_PUBKEY_ENCRYPT	X	X
CGX_PUBKEY_DECRYPT	X	X
CGX_IMPORT_PUBKEY	X	X
CGX_EXPORT_PUBKEY	X	X
Digital Signature Commands		
CGX_SIGN	X	X
CGX_VERIFY	X	X
Hash Commands		
CGX_HASH_INIT	X	X
CGX_HASH_DATA	X	X
CGX_HASH_ENCRYPT	X	X
CGX_HASH_DECRYPT	X	X
Prf Commands		
CGX_PRF_DATA	X	X
CGX_PRF_KEY	X	X
CGX_MERGE_KEY	X	X
CGX_MERGE_LONG_KEY	X	X
CGX_LONG2BLACK	X	X
Math Commands		
CGX_MATH	X	X

Table 3

Command Descriptions:

CGX_INIT initializes the CGX Kernel and CryptIC hardware, runs a set of basic self-tests, and allows the caller to configure two classes of configuration settings:

- Increase the default number of Key Cache Registers (from 15 up to 700)
- Specify various configuration parameters associated with the CGX Kernel (via the Kernel Configuration String)

CGX_DEFAULT initializes the CGX Kernel and/or CryptIC hardware, and restores application-definable settings to factory defaults. This command is typically used to reset any customized settings which may have previously been selected using CGX_INIT.

CGX_RANDOM gets bytes of random data from the pseudo random number generator.

CGX_GET_CHIPINFO provides information about the secure kernel and the CryptIC, including the revision level of the hardware and CGX firmware, the current settings of the Program Control Data Bits (PCDBs), and the chip's serial number.

CGX_ZEROIZE_KEYS is used to delete all of the KCRs including the LSV from KCR 0. Furthermore, it exits from the CGX library .

CGX_SELF_TEST initializes and tests the CryptIC and the CGX kernel. The Self Test command restores the CGX kernel to factory defaults upon completion. If the application has customized the CGX kernel using the KCS CGX_INIT must be run again to restore application-definable settings.

CGX_GEN_KEK generates an internal key encryption key using the CGX's pseudo random number generator and places it into the specified Key Cache Register.

CGX_GEN_KEY generates a symmetrical key using the CGX's pseudo random number generator and places it into the specified Key Cache Register. Optionally, the newly generated key may be returned to the caller in a Black (DES or TDES encrypted) form. The random key bits are transformed into the secret key form as directed by the type of secret key specified in the argument interface.

CGX_DERIVE_KEY (non-FIPS compliant) allows a user secret key to be created from an application's pass-phrase. The secret key is derived by taking the one-way Hash of the application's pass phrase and using the resulting message digest as the secret key bits. The 'raw' message digest bits are transformed into the secret key form as directed by the type of secret key specified (i.e. key_type) in the argument interface and placed into the specified Key Cache Register.

CGX_TRANSFORM_KEY allows a user supplied black secret key into a hash digest to be used as a precompute in the PRF functions or in an HMAC operation.

CGX_MERGE_KEY takes key material from two secret keys and combines the material to form a third secret key. The key material in two input keys, key1 and key2, is combined in a caller-specified way. The possible combine operations are <u>concatenate</u>, <u>exclusive-or</u>, and <u>hash</u>. The resulting material (or the leading bytes of the resulting material, if the resulting material is more than needed to create the new key) becomes the key material for a new key. Three or more input keys may be combined by merging the output of one merge_key operation with yet another input key, and repeating this step as often as necessary.

CGX_UNCOVER_KEY decrypts the Black secretkey, bk, to a Red form and places it into the key cache register (KCR) indicated by the input argument, destkey. A Black secret key is defined as a key stored in IRE internal format (which has therefore been encrypted and authenticated with a keyed hash). This allows an application to securely store Black secret keys outside of CGX for later use by the CGX kernel.

CGX_LOAD_KEY is used to load a plaintext user secret key into a specified Key Cache Register. The secret key to be loaded is in the Red form. Depending on the value of the use argument, the key can be used as either a KEK or as a DEK. This key is known as a user key to the CGX Kernel and can never be covered by the LSV (the CGX Kernel does not allow it).

CGX_EXPORT_KEY allows the application to move an IRE internal secret key form into an External secret key form. The External secret key form must be covered either with a secret key or public key, this is specified by the application via the command arguments.

CGX_IMPORT_KEY allows the application to load and create an IRE internal secret key from an External secret key form.

CGX_LOAD_KG is used to load a DES/Triple DES secret key into the hardware crypto-engine (i.e. KG or key generator) or an RC5 key into the RC5 software key generator (supported in the software CGX kernel model only). The typical use of this command is to fully optimize secret key traffic by pre-loading the traffic key in advance or for loading a different DKEK into the DKEK register of the hardware crypto-engine. In FIPS mode, this service is not valid for the software CGX module.

CGX_DESTROY_KEY command is used to remove a secret key from the specified key cache register.

CGX_ENCRYPT is used to perform the symmetrical encryption of plain-text data and return the cipher-text to the application in the specified buffer.

CGX_DECRYPT is used to perform the symmetrical decryption of cipher-text data and returning the plain-text to the application in the specified buffer.

CGX_HASH_INIT (Initialize Hash) is used to initialize a Hash context block (data structure type hash_cntxt.) The command is used in preparation for a Hash function computation. After initialization, the Hash context block may subsequently be used as a parameter to a sequence of one or more CGX operations, such as CGX_HASH_DATA, which perform the Hash computation. At any given time, an application may have several separate independent hash computations in various stages of completion. Each hash computation will have its own dedicated hash context; each context contains the current state information of its corresponding Hash computation. The computation types supported are SHA-1 and MD5 one way Hash algorithms. Both Hash algorithms have a limit of $2^{64} - 1$ bits cumulative input data length. Upon completion of this operation, the hash context will contain a NULL value in the digest member of the hash_cntxt object (since the hash isn't 'closed'). When the hash computation is completed and the context is closed, the digest member will contain a valid hash digest: i.e., the result of the hash computation.

CGX_HASH_DATA (Hash Data) is used to calculate a Hash value over data supplied by the calling application. The hash value is computed over a stream of data octets (8-bit data bytes) which optionally may begin with a key whose octets are treated as data to be hashed (thus creating a 'keyed hash'), then may include a virtually unlimited number of non-key data octets and optionally concludes with a trailing key whose octets are also treated as data to be hashed. If both leading and trailing data keys are included in the hashed data stream, they may be the same or different. For security reasons, a key may not be inserted into the middle of the data being hashed.

CGX_HASH_ENCRYPT (Hash and Encrypt) is used to perform both a hash computation and a symmetrical encryption of a data buffer. In a single call, the invoking application can encrypt a block of data and simultaneously compute a hash function over the data block. The hash can be computed over the input data <u>before</u> encryption or over the resulting data <u>after</u> encryption.

CGX_HASH_DECRYPT (Hash and Decrypt) operation is nearly identical to the CGX_HASH_ENCRYPT operation. The essential difference is that this command uses the key referred to in the crypto context parameter to perform a symmetric decryption, not an encryption. Typically, CGX_HASH_DECRYPT is used to decrypt a message and also compute the message digest. This recovers the original plaintext and the message digest computed by a CGX_HASH_ENCRYPT command. For this operation to be the logical inverse of a CGX_HASH_ENCRYPT operation, all parameters to both operations should be logically equal, except the order parameter, which should be reversed. (HASH-THEN-DECRYPT is the inverse of ENCRYPT-THEN-HASH.) Some variance is naturally permitted within the term *logically equal*. For example, the keys must be equal, but can reside in different KCRs and the key

load options may, of course, vary. The message data input to HASH_DECRYPT must have been produced by HASH_ENCRYPT, but the blocking into 64-bit–multiple segments may vary from that used in the encryption.

CGX_PRF_DATA hashes one, two or three data items, of different types, into the inner hash of an HMAC being generated. The items (in the order they are processed) are:

- a secret key (specified by argument secretkey *bk)
- a g^xy DH shared key specified in argument publickey *gxypk

RED data (specified in argument (VPTR)*dptr of a specified number of bytes (bytecount.)

CGX_PRF_KEY can be used to complete the IPsec HMAC. Command arguments supply two open hash contexts known as the inner hash context and the outer hash context, both of which are covered. (Additional arguments supply the crypto contexts needed to uncover the hash contexts.) The command closes the inner hash context (its internal copy of the inner hash context – the caller's copy is not affected.) Then it hashes the digest of the inner hash context into the outer hash context. Then it closes the outer hash context (its copy of the outer hash context) and creates a secretkey of type specified by the caller from the outer hash digest and returns the key, covered, to the caller. It also leaves the created key in a specified key cache register, ready to use for encryption.

CGX_MERGE_LONG_KEY is quite similar to the CGX_MERGE_KEY command. The essential difference is that the output key created by CGX_MERGE_LONG_KEY is not a data encryption key; rather it is merely a long key that can be used subsequently (for example by command CGX_EXTRACT_LONG) to create encryption keys. The output data type of CGX_MERGE_LONG_KEY is a container, not a true key; it is perhaps misnamed as a **longkey** data type. A variable of this type can hold up to 64 bytes of key information. Such a data type provides intermediate storage, for example, for the 48 bytes resulting from concatenating two 24 byte keys, which then can be used (by CGX_EXTRACT_LONG) to produce an encryption key from the middle 24 bytes of the concatenation. The CGX_MERGE_LONG_KEY command takes key material from two keys and combines the material to form a new long key. The first input key, key1, may be either an ordinary encryption key (type secretkey) or a longkey. The second input key, key2, must be an ordinary encryption key. The key material in two input keys, key1 and key2, is combined in a caller-specified way. The possible combine operations are concatenate, exclusive-or, or hash. The resulting material becomes the key material for the new key. Three or more input keys may be combined by merging the output of one merge_long_key operation with yet another input key. One caveat to be observed is that when the concatenate operation is requested, the user must ensure that the sum of the two lengths of the input keys does not exceed the 64-byte maximum length of a long key.

CGX_EXTRACT_LONG_KEY creates a secret key from key material supplied within a longkey.

CGX_GEN_PUBKEY will generate an entire public keyset comprised of the modulus, private, and public blocks. This operation can create public keysets for several public key algorithms. This interface is over-loaded and currently supports Diffie-Hellman, RSA, and DSA public keys. The returned keyset will consist of data stored in little endian order.

CGX_GEN_NEWPUBKEY is used to generate new public and private blocks for a Diffie-Hellman or DSA public keyset. This command is only valid for Diffie-Hellman or DSA public keysets. The command allows the flexibility to import a public key block from the application and use it to generate the new private and public blocks. The application has control over which parts to generate and return via the two control constants CGX_X_V (the private part) and CGX_Y_V (the public part). Using combinations of these control masks allows the application with a flexible key generation interface.

CGX_GEN_NEGKEY will complete the Diffie-Hellman exchange by deriving the shared key from the receiver's public key. CGX supports dynamically negotiated keys as specified in the X9.42 Standard. This command also supports the generation of a g^xy key blob. The key blob can be used as a component to the creation of IPsec operations. This command is only used for Diffie-Hellman public keysets.

CGX_EXPORT_PUBKEY allows the application to move an IRE public keyset form into an external public key form. The external form must be covered with a KEK, this is specified by the application via the command arguments.

CGX_IMPORT_PUBKEY allows the application to move an external public key back into CGX in the IRE public keyset form. The external form must be covered either with a secret key or public key, this is specified by the application via the command arguments.

CGX_PUBKEY_ENCRYPT is used to encrypt the application's data using the RSA encryption algorithms. This operation also may be used to perform RSA signature verification using the public key component of a public keyset.

CGX_PUBKEY_DECRYPT is used to decrypt the application's data using the RSA encryption algorithms. This operation also may be used to perform RSA signing using the private key of a public keyset.

CGX_SIGN command is used to sign the application's message or message digest using the DSA digital signature algorithm.

CGX_VERIFY is used to verify the signature of the application's message using the DSA public key algorithm.

CGX_MATH is a set of cgx commands that perform various mathematical functions.

D. Security Rules

This section documents the security rules enforced by the cryptographic module to implement the security requirements for both the FIPS 140-1 Level 1 and Level 2 module except as noted.

1. *The cryptographic module shall provide two distinct operator roles. These are the User Role, and the Cryptographic Officer Role.*

2. *(Level 1 only) When the cryptographic module is installed on Windows NT, the operating system shall be configured for single user mode.*

3. *(Level 2 only) The cryptographic Module shall provide role-based authentication via the Windows NT 4.0 authenticated login process as tested on a Compaq Deskpro 6400. Windows NT 4.0 on a Compaq Deskpro 6400 is C2 equivalent tested under ITSEC.*

4. *When the module has not been properly initialized, the operator shall not have access to any cryptographic services and CGX will remain in the Not Loaded Sate.*

5. *Upon the application of power or when commanded by the operator, the cryptographic module shall perform the following tests:*

   a) *PRAM Test*
   b) *Pseudo Random Number Generator Statistical Test*
   c) *DES Encryption/Decryption Algorithm Known Answer Test*
   d) *Public Key Encryption Algorithm Key Pair Test*
   e) *SHA-1 Algorithm Known Answer Test*
   f) *DSA Algorithm Known Answer Test*
   g) *Diffe-Helman Known Answer Test*
   h) *RSA Known Answer Test*
   i) *Triple DES Encryption/Decryption Algorithm Known Answer Test*

6. *At any time the module is in an idle state, the operator shall be capable of commanding the module to perform the power-up self test.*

7. *CGX utilizes the following cryptographic and hashing algorithms:*

Symmetrical

	DES	*FIPS 81 and FIPS 46-3*
		Electronic Code Book (ECB)
		Cipher Block Chaining (CBC)
		64-bit Output Feedback (OFB)
		64-bit Cipher Feedback (CFB)
	TDES	*FIPS 46-3*
		Electronic Code Book (ECB)
		Cipher Block Chaining (CBC)
		64-bit Output Feedback (OFB)
		64-bit Cipher Feedback (CFB)
	RC5	

Asymmetrical

	DSA	*FIPS 186-1*
	RSA	
		encrypt/decrypt
		signatures

Diffie-Hellman

Oneway HASH

MD2
MD5
SHA-1 FIPS 180-1
RIPEMD-128
RIPEMD-160

8. *Prior to each use, the internal Random Number Generator shall be tested using the Conditional test specified in FIPS 140-1 §4.11.2 paragraph 5.*

9. *The CGX cryptographic module must always be properly initialized prior to it being used. If an operator attempts to execute a CGX command without first executing the CGX_Init command, then CGX will automatically execute CXG_INIT on its own prior to processing the requested command.*

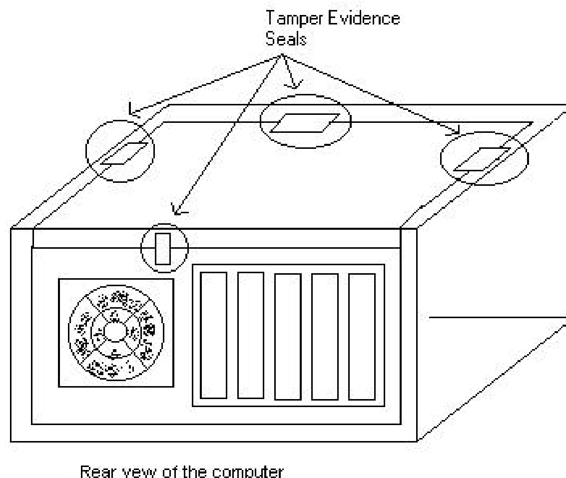
10. *Unencrypted (Red) keys can never be returned by CGX. All keys passed back to the caller are always encrypted under a higher level KEK*

11. *Applications utilizing the CGX cryptographic module must conform to the requirements in FIPS 140-1. It is the responsibility of the application not of the CGX cryptographic module to handle red key entry.*

12. *The CGX cryptographic module was written in a high level language.*

13. *(Level 2 only) The module shall reside in an opaque enclosure and shall be protected by tamper evident seals. The tamper evident seals will be applied to the case interfaces. These seals have integral press type adhesive, which is immediately effective and fully cures in less than 2 hours. It is the operator's responsibility to apply the labels and to verify their integrity while in use. Application of the tamper evidence labels shall be performed by removing the seals from the carrier and applying it on a clean, dry surface area. It is recommended that both the seals and the surface area temperature be relatively warm (i.e. room temperature) during application. Contact IRE for acquisition information concerning the tamper evidence seals.*

    The diagram below identifies the appropriate placement of seals at case interface locations:



Rear vew of the computer

E. Definition of Security Relevant Data Items

SYMMETRIC KEYS

a) Data Encryption Key (DEK): This is a DES or Triple-DES key used to encrypt user traffic.

b) Key Encryption Key (KEK): This is a DES or Triple-DES key used only to encrypt other keys.

c) Generator Key Encryption Key (GKEK): This is a special Triple-DES key used only to encrypt other keys, and is itself protected by the Local Storage Variable (LSV).

d) Local Storage Variable (LSV): This is a unique Triple-DES key used as the root key to recover other keys after a power outage. The LSV is always loaded into Key Cache Register #0. It cannot be exported from CGX. The LSV is stored encrypted within the Windows Registry.

ASYMMETRIC KEYS

a) Public Key: This is the public component of an RSA, DSA or Diffie-Hellman key pair.

b) Private Key: This is the private component of an RSA, DSA or Diffie-Hellman key pair.

OTHER OBJECTS

a) Initialization Vector (IV): This is a 64 bit random number used to initialize the DES encryption algorithm. Each algorithm is initialized with a unique IV, supplied by the application or from the PRNG, for each message encrypted.

b) Kernel Configuration String (KCS): This is a configuration string that sets-up certain features of the CGX kernel during the Initialization process. Two of the relevant configuration options are:

- Enable FIPS 140-1 compliant RNG. This parameter turns on the ANSI X9.17 randomizer which is applied to the random number entropy source, the X9.17 seed key only resides in RAM, presented to CGX. This feature must be enabled for the FIPS 140-1 compliant version of CGX.
-

c) Key Attribute Bits: This is a bit-mapped field which is attached to any key and specifies its Type and Use. The key type specifies whether the key is a DEK, KEK, etc..

d) Key Cache Register (KCR): This is a volatile key storage house for a fixed number of secret keys. The volatile key area is also referred to as the actively working keys. All cryptographic commands operate only on the active volatile working keys.

G. Service to SRDI Access Operation

	SRDI									
User Service	DEK	KEK	GKEK	LSV	PublicKey Public Component	PublicKeyPrivate Component	IV	KCS	KeyAttribute Bits	KCR
CGX_INIT				R				R		W
CGX_DEFAULT				R				R		W
CGX_RANDOM										
CGX_GET_CHIPINFO										
CGX_ZEROIZE_KEYS										D
CGX_SELF_TEST	RW							RW		RW
Symmetric Key Commands										
CGX_UNCOVER_KEY	RW	RW	RW				RW		RW	RW
CGX_GEN_KEK		RW		R			RW		RW	RW
CGX_GEN_KEY	RW	RW	R				RW		RW	RW
CGX_LOAD_KEY	RW	RW	R				RW		RW	RW
CGX_DERIVE_KEY	RW	RW	R				RW		RW	RW
CGX_TRANSFORM_KEY	RW	RW					RW		RW	R
CGX_EXPORT_KEY	M	M					RW		R	R
CGX_IMPORT_KEY	M	M					RW		W	R
CGX_DESTROY_KEY										D
CGX_LOAD_KG									R	R
CGX_ENCRYPT	R						RW		R	R
CGX_DECRYPT	R						RW		R	R
Asymmetric Key Commands										
CGX_GEN_PUBKEY		R	R		W	RW	R		R	R
CGX_GEN_NEWPUBKEY		R	R		W	RW	R		R	R
CGX_GEN_NEGKEY	RW	RW			R	R	R		R	R
CGX_PUBKEY_ENCRYPT		R			R	R	R		R	R
CGX_PUBKEY_DECRYPT		R			R	R	R		R	R
CGX_IMPORT_PUBKEY		R				M	R		R	R
CGX_EXPORT_PUBKEY		R				M	R		R	R
Digital Signature Commands										
CGX_SIGN		R	R		R	RW	R			R
CGX_VERIFY		R	R		R		R			R
Hash Commands										
CGX_HASH_INIT										
CGX_HASH_DATA										
CGX_HASH_ENCRYPT	R						RW			R
CGX_HASH_DECRYPT	R						RW			R
Prf Commands										
CGX_PRF_DATA	R	R	R		R	RW	R			R
CGX_PRF_KEY	RW	RW	R		R	RW	R			R
CGX_MERGE_KEY	RW	R	R				R			R
CGX_MERGE_LONG_KEY	RW	R	R				R			R
CGX_LONG2BLACK	RW	RW	R				R			R
Math Commands										
CGX_MATH										

R = Read W = Write M = Modify D = Delete

Table 4

H. List of Acronyms

Acronym	Description
API	Application Programming Interface
CGX	**C**rypto**G**raphic e**X**tensions
DEK	Data Encryption Key
DES	Data Encryption Standard
D-H	Diffie-Helman
DSA	Digital Signature Algorithm
EMC	Electromagnetic Compatibility
EMI	Electromagnetic Interference
FIPS	Federal Information Processing Standard
GKEK	Generator Key Encryption Key
HMAC	Hash Message Authentication Code
IPsec	Internet Protocol Security
IRE	Information Resources Engineering, Inc.
ITSEC	Information Technology Security Evaluation Criteria
IV	Initialization Vector
KCR	Key Cache Register
KCS	Kernel Configuration String
KEK	Key Encryption Key
KG	Key Generator
LSV	Local Storage Variable
MD5	Message Digest 5
OS	Operating System
PC	Personal Computer
PCDB	Program Control Data Bit
PRAM	Program Random Access Memory
PRF	Pseudo Random Function
PRNG	PseudoRandom Number Generator
RAM	Random Access Memory
RNG	Random Number Generator
RSA	Rivest Shamir Adleman
SHA-1	Secure Hash Algorithm