CGX Cryptographic Module
Software Version 3.18

Security Policy
Level 1

Security Rule Specification

Approved Document

Revision: 1.2
Date: 20 February 2004
Status: Approved

SafeNet, Inc. maintains a website with up-to-date technical documentation for our customers. Contact SafeNet for access:

www.safenet-inc.com

For Further Information:

siliconsales@safenet-inc.com

SafeNet, Inc.
100 Conifer Hill Drive, Suite 513
Danvers, MA 01923
U.S.A.
Phone: (+1) 978-539-4800 Fax: (+1) 978 739-5698

SafeNet BV
The Netherlands
Boxtelseweg 26A
5261 NE Vught
P.O. Box 22
5260 AA Vught
The Netherlands
Phone: +31-73-6581900 Fax: +31-73-6581999

SafeNet, Inc.
Corporate Headquarters
8029 Corporate Drive
Baltimore MD, 21236
Phone: (410) 931-7500 Fax: (410) 931-7524

Revision History

Rev	Page(s)	Date	Author	Purpose of Revision
0.1	All	3/5/03	LG	Initial submission
0.2	All	5/12/03	LG	Updated submission for review
0.3	All	6/5/03	LG	Updated submission for review
1.0	All	6/16/03	LG	Approved by Rick DeFelice
1.1	All	7/16/03	LG	Modifications per NIST review
1.2	All	2/20/04	DP	Modifications per NIST/CSE review

TABLE OF CONTENTS

1 Overview

The SafeNet CGX (**C**rypto**G**raphic e**X**tensions) software version 3.18 library is a suite of approximately 143 cryptographic functions, which are available to applications which require security services. It is currently implemented on all production grade Windows operating systems (9.x, NT 4.0, 2000, XP) as well as embedded in numerous SafeNet cryptographic accelerator chips. For the purposes of the FIPS 140-2 validation, the scope of the of the product being submitted for validation has been limited to 40 kernel functions, implemented as a multi-chip standalone module entirely in software as either a dynamic link library (IreCGX.dll) or as a kernel service (crypto.sys/crypto.vxd) on the above listed Windows operating systems.

2 CGX Functional Specifications

The SafeNet security software which is made available to applications running on a Windows 9.x, NT 4.0, 2000, or XP platform is designated the SafeNet CGX Kernel. 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.

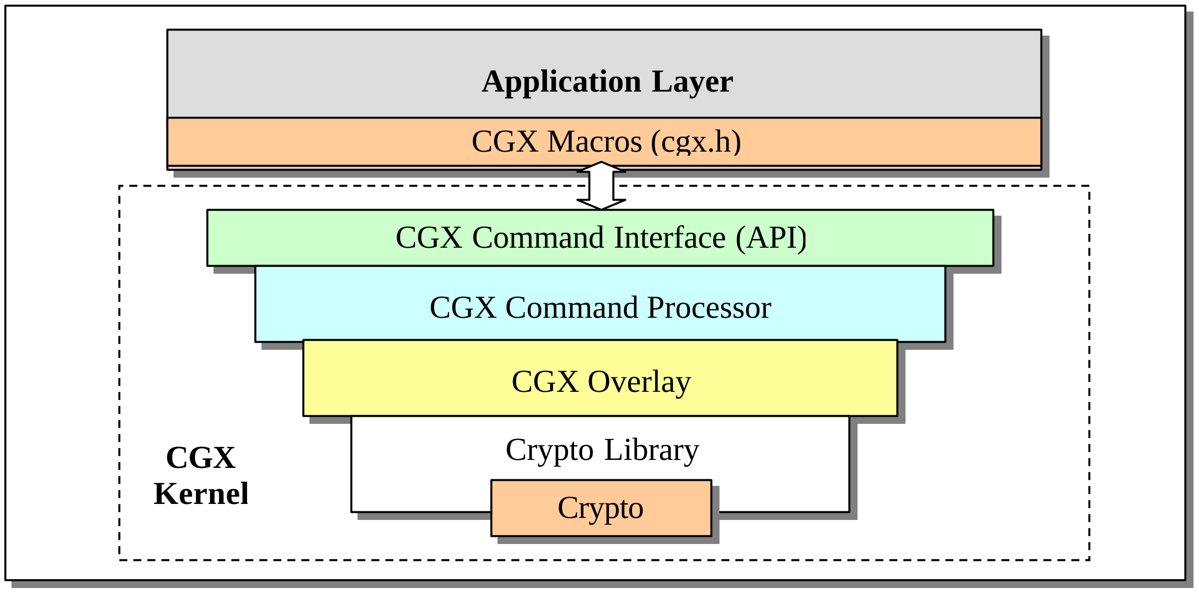

Figure 1 *CGX layers*

To execute a CGX command, a structure is populated with arguments and a 'C' language call is made to the CGX Kernel, passing a pointer to the structure. Alternatively, a macro from the header file cgx.h may be used with the appropriate arguments for the command. The macro for the command will insert the arguments into the control block structure and invoke the CGX Kernel.

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 sections below.

2.1 APPLICATION LAYER

The application layer is where the actual application program and data space reside. The application can implement anything from a router security co-processor to a V.34 mo dem data pump, but for purposes of this FIPS 140-2 validation the application is assumed to be a software-only application.

In order to access the cryptographic services, the application must invoke the CGX command interface and supply a command code and arguments. It is likely that the application layer will include a '*CGX Processing Manager*' which accepts host-originated requests, formats them, and then issues the call to CGX for processing.

Residing as part of the application layer are the macro functions that SafeNet provides in its cgx.h and ecgx.h files. These optional macros assist the application in preparing the command messages prior to calling the CGX Kernel.

2.2 CGX COMMAND INTERFACE LAYER

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

The software interface to the CGX Kernel is via 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 that the application can read to determine the result of a requested cryptographic service. The *command block* is used to request a specific cryptographic command and to provide a means of supplying arguments.

Therefore, all communications between the application and the CGX Kernel is via the command interface and a *kernel block* and *command block*. The command interface is discussed in more detail in Chapter 3 of the CGX Programmers Guide.

2.3 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.

2.4 CGX OVERLAY LAYER

The CGX overlay layer is provided as the interface into SafeNet'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 a platform unmodified.

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

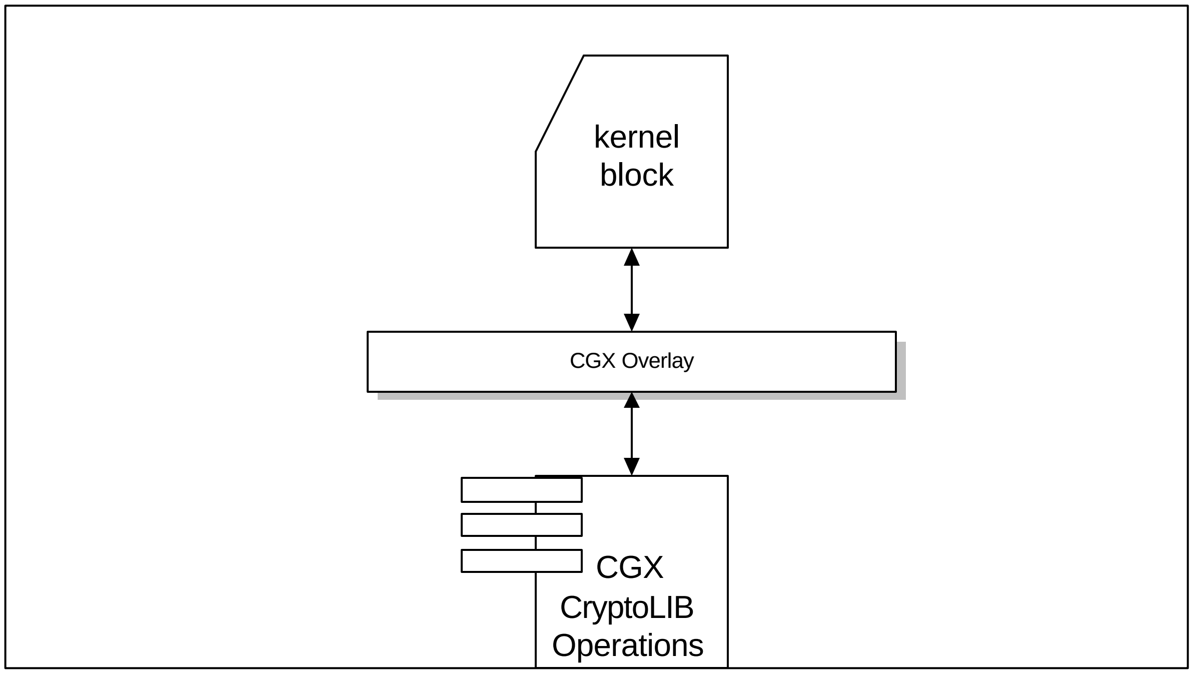

Figure 2 *CGX overlay interface*

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.

2.5 CRYPTOLIB LAYER

The CryptoLIB layer contains SafeNet's Crypto Library software. The CryptoLIB software is a library of many cryptographic classes implementing various cryptographic algorithms including symmetrical encryption algorithms, one-way hash functions, and public key operations.

2.6 CRYPTOGRAPHIC BOUNDARY

Figure 3 illustrates the cryptographic boundary of the CGX module, interconnections among major components of the CGX module and between the CGX module and equipment or components outside of the cryptographic boundary. For the purpose of completeness, the Hardware Provider Interface (HPI) is included inside of the cryptographic boundary; however, since it's purpose is to provide communication between the CGX kernel and an installed cryptographic accelerator chip, for the purposes of this validation the only function performed by the HPI is to confirm the absence of hardware acceleration.

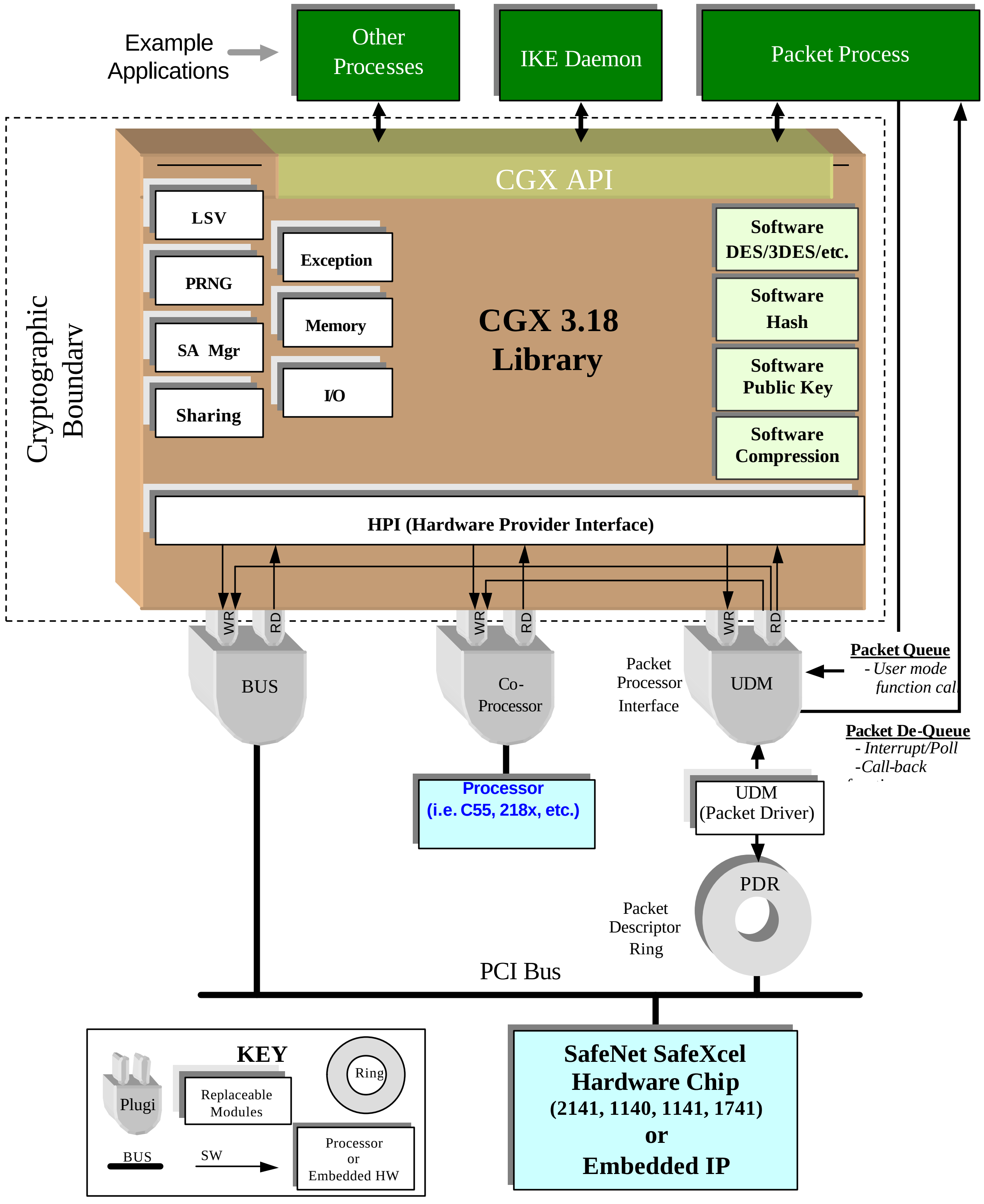

Figure 3 *Definition of Cryptographic Boundary*

3 Security Level

The cryptographic module meets the overall requirements applicable to Level 1 security of FIPS 140-2 when running on any production grade Windows-based operating system and commercial desktop Intel-based computer.

Table 1 *Module Security Level 1 Specification*

Security Requirements Section Level	
Cryptographic Module Specification	1
Module Ports and Interfaces	1
Roles, Services and Authentication	1
Finite State Machine	1
Physical Security	N/A
Operational Environment	1
Cryptographic Key Management	1
EMI/EMC	3
Self Tests	1
Design Assurance	3
Mitigation of Other Attacks	1

4 Roles and Services

The CGX cryptographic module does not support physically distinct operator roles, as any authentication is performed external to the cryptographic boundary. There is no separation of functionality for a crypto user and crypto officer, rather the distinction lies in the function that is being executed. When the calling application is performing functions such as initialization, zeroize keys and self-test, it is acting as a virtual crypto officer; when performing operations such as encrypt, decrypt, etc. is acting a virtual crypto user.

Role	Type of Authentication	Authentication Data
User Role	None	N/A
Cryptographic Officer Role	None	N/A

Table C1. *Roles and Required Identification and Authentication*

Authentication Mechanism	Strength of Mechanism
None	N/A

Table C2. *Strengths of Authentication Mechanisms*

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

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

Table C3. *Services Authorized for Roles*

4.1 CGX Kernel Command Descriptions

The CGX module interface in it's simplest form consists of the kernel block, the command block and a single module interface, **cgx_transfer_secure_kernel**. Based on how the kernel and command blocks are populated when **cgx_transfer_secure_kernel** is invoked, the CGX kernel will perform the requested function and return output as appropriate. To aid the programmer in accessing CGX functionality, the CGX library is packaged with a set of macros that will populate the command and kernel blocks appropriately and call **cgx_transfer_secure_kernel**. These macros can be used for direct access to the user-mode implementation of CGX SE, IreCGX.dll.

Access to the kernel-mode implementation of CGX (crypto.sys for NT-based installations and crypto.vxd for 9.x installations) is provided via the SafeNet CSP (**C**ryptographic **S**ervice **P**rovider), which uses the Microsoft-defined API CryptoAPI. CryptoAPI is a standard suite of API calls that allow applications to access cryptographic functionality provided by Microsoft and other vendors' CSPs. Using CryptoAPI, the application can indicate which CSP it wants to use and the cryptographic function it would like to perform such as encrypt, decrypt, etc. The CSP acts as the translator between the CryptoAPI call and the vendor-implemented cryptographic function to be performed. More information on using the CryptoAPI can be obtained on the Microsoft MSDN web site (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/security/security/cryptography_reference.asp).

The following is a brief description of the functionality available within the CGX module, referenced by the SafeNet-defined macro names.

CGX _INIT initializes the CGX Kernel, 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 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, including the revision level of the hardware and CGX firmware and the current settings of the Program Control Data Bits (PCDBs).

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 SE's pseudo random number generator and places it into the specified Key Cache Register.

CGX _GEN_KEY generates a symmetrical key using the CGX SE'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 SafeNet 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 SafeNet 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 a SafeNet 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 exc eed the 64-byte maximum length of a long key.

CGX _EXTRACT_LONG_KEY creates a secret key from key material supplied within a longkey object. The argument *key1* is a longkey object supplied in the black along with a crypto_cntxt object to be used to uncover it. Bytes are extracted from *key1* starting at *offset* for *length* bytes. The operation will fail if the input longkey does not contain *length*+*offset* bytes. A secret key, *bk*, is created from this key material using the *type* and *use* arguments. The secret key is covered using *bk_kek*, and returned to the application.

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 ANSI 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 SafeNet 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 SafeNet 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.

4.2 Security Rules

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

1. The cryptographic module shall provide two distinct operator roles by virtue of the type of operation being performed. These are the User Role, and the Cryptographic Officer Role.

2. The cryptographic Module does not provide authentication.

3. When the module has not been properly initialized, the operator shall not have access to any cryptographic services and CGX will remain in the Error State.

4. Upon the application of power or when commanded by the operator, the cryptographic module shall perform the following tests:
   Triple DES Encryption/Decryption Algorithm Known Answer Test
   DES Encryption/Decryption Algorithm Known Answer Test
   SHA-1 Algorithm Known Answer Test
   AES Encryption/Decryption Known Answer Test
   HMAC Known Answer Test
   Pseudo Random Number Generator Known Statistical Test
   Pseudo Random Number Generator Known Answer Test
   DSA Known Answer and Public Key Pair Tests
   RSA Known Answer and Public Key Pair Tests
   Diffie-Hellman Known Answer Test
   Software/firmware integrity check

5. 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.

6. CGX utilizes the following cryptographic and hashing algorithms:
   FIPS-Approved:

Cert #72	DES	FIPS 81 and FIPS 46-3 (For use with legacy systems only) Electronic Code Book (ECB) Cipher Block Chaining (CBC) 64-bit Output Feedback (OFB) 64-bit Cipher Feedback (CFB)
Cert#11	TDES	FIPS 46-3 Electronic Code Book (ECB) Cipher Block Chaining (CBC) 64-bit Output Feedback (OFB) 64-bit Cipher Feedback (CFB)
Cert#75	AES	FIPS 197 128, 192 and 256 bit keys Electronic Code Book (ECB) Cipher Block Chaining (CBC)
Cert#30	DSA	FIPS 186-1
Cert#30	SHA-1	FIPS 180-1

Cert#30;Vendor Affirmed	HMAC-SHA-1	FIPS 198

Non FIPS-Approved

RC5	
RSA	encrypt/decrypt
	signatures
Diffie-Hellman	
MD2	
MD5	
RIPEMD-128	
RIPEMD-160	

7. Prior to each use, the internal Random Number Generator shall be tested using the Conditional test specified in FIPS 140-2 §4.9.1.

8. Prior to each use, the binary files crypto.sys/crypto.vxd and IreCGX.dll shall perform a software/firmware integrity check via a HMAC/SHA-1 hash of the binary to be compared with the digest created at build time. If the two digests are not equal, the module shall enter an error state and further use prevented.

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 140-2. 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 the C high level language.

13. To operate the CGX module in FIPS mode, the following must be observed:

- The operating system must be in single user mode.
- Only the functions listed in section 4.1, CGX Kernel Command Descriptions should be used.
- Only the algorithms listed above as FIPS-approved should be used.
- The initialization block must be configured with an external HostKcrCache for 255 KCRs, and a SA_Configuration setting of 1.
- The Kernel Configuration String must contain the following settings:

```
flipsha = CGX_FLIP_SHA1_FINAL
flipgxy = TRUE
fips140_1 = TRUE
fips_enable = (CGX_FIPS_ENABLE_ALL_LOWER &
~CGX_FIPS_BIST_ENABLE_PRAM) |
CGX_FIPS_ENABLE_STRONG_PRNG |
CGX_FIPS_ENABLE_REDKEY |
CGX_FIPS_ENABLE_PAIRWISE |
CGX_FIPS_LOAD_ALTERNATE_LSV
fips_enable_upper = CGX_FIPS_ENABLE_ALL_UPPER
```

4.3 Definition of Security Relevant Data Items (FIPS and non-FIPS)

4.3.1 SYMMETRIC KEYS

- Data Encryption Key (DEK): This is a DES, Triple-DES or AES key used to encrypt user traffic.
- Key Encryption Key (KEK): This is a DES, Triple-DES or AES key used only to encrypt other keys.
- 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).
- 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 SE. The LSV is stored encrypted within the Windows Registry using TDES and a hard-coded key.
- HMAC Key: This is the key used in HMAC-SHA-1.

4.3.2 ASYMMETRIC KEYS

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

4.3.3 OTHER OBJECTS

- 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.
- 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 140-2 compliant RNG. This parameter turns on the ANSI 9.31 A.2.4 randomizer which is applied to the random number entropy source, the ANSI 9.31 A.2.4 seed key only resides in RAM, presented to CGX. This feature <u>must</u> be enabled for the FIPS 140-2 compliant version of CGX.
  - Allows the Crypto Officer to enable/disable red key parity checking from the Kernel Configuration String.
- 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.
- 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.

4.4 Service to SRDI Access Operation

	SRDI									
User Service	DEK	KEK	GKEK	LSV	PublicKey Public Component	PublicKey Private 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_EXTRACT_LONG_KEY	RW	RW	R				R			R
Math Commands										
CGX_MATH										

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

Table C4. *Access Rights within Services*

5 Physical Security

For level 1 FIPS 140-2 validation, CGX may be installed on any commercially available, production grade Intel based computer, and therefore physical security will vary based on model and manufacturer. In general these features include an opaque enclosure and may also include tamper-evident seals. No physical requirements are necessary but it is recommended those responsible for the Cryptographic Module inspect the physical platform housing the module routinely.

6 Mitigation of Other Attacks

Other Attacks	Mitigation Mechanism	Specific Limitations
Kocher Timing Analysis Attack	Check for attack during Diffie-Helman key generation.	None.

Table C5. *Mitigation of Other Attacks*

The Kocher Timing Analysis Attack is a timing attack theory developed by Paul Kocher, President of Cryptography Research Inc. The basis of the theory is that by carefully measuring the amount of time required to perform private key generation, attackers may be able to find the fixed Diffie-Hellman exponents, factor RSA keys, and break other cryptosystems. CGX mitigates this type of attack by always performing the mu ltiply in the inner square loop of exponentiation which results in constant time, and has implemented RSA blinding for RSA decryption.

Appendix A List of Acronyms

Acronym	Description
AES	Advanced Encryption Standard
API	Application Programming Interface
CGX SE	**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
ITSEC	Information Technology Security Evaluation Criteria
IV	Initialization Vector
KAT	Known Answer test
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
SRDI	Security Relevant Data Items
SHA-1	Secure Hash Algorithm