



## Thales nShield HSM Security Policy

nShield F3 10+, nShield F3 500+,  
nShield F3 6000+,  
nShield F3 500+ for nShield Connect+,  
nShield F3 1500+ for nShield Connect+,  
nShield F3 6000+ for nShield Connect+  
in level 2 mode

---

**Version:** 3.3

**Date:** 03 November 2015

Copyright 2015 Thales UK Limited. All rights reserved.

Copyright in this document is the property of Thales UK Limited.

This document is non-proprietary. Reproduction is authorised provided the document is copied in its entirety without modification and including this copyright notice.

Words and logos marked with ® or ™ are trademarks of Thales UK Limited or its affiliates in the EU and other countries.

Information in this document is subject to change without notice.

Thales UK Limited makes no warranty of any kind with regard to this information, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Thales UK Limited shall not be liable for errors contained herein or for incidental or consequential damages concerned with the furnishing, performance or use of this material.

### **Patents**

UK Patent GB9714757.3. Corresponding patents/applications in USA, Canada, South Africa, Japan and International Patent Application PCT/GB98/00142.

# Versions

To support the range of nShield hardware platforms, multiple variants of this document are generated from the same source files.

| Version | Date              | Comments  |
|---------|-------------------|---|
| N/A     | 13 August 1998    | nFast nF75KM and nF75CA SCSI modules f/w 1.33.1   |
| N/A     | 18 January 2000   | nForce and nShield SCSI and PCI modules f/w 1.54.28   |
| N/A     | 20 December 2000  | nForce and nShield SCSI and PCI modules f/w 1.70  |
| N/A     | 1 March 2000      | nForce and nShield SCSI and PCI modules f/w 1.70  |
| 1.0.7   | 23 May 2001       | nForce and nShield SCSI and PCI modules f/w 1.71<br>Adds SEE  |
| 1.0.9   | 14 September 2001 | nForce and nShield SCSI and PCI modules f/w 1.71.91<br>Adds Remote Operator Card Sets, Foreign Token Access, Feature Enablement |
| 1.1.25  | 6 May 2002        | nForce and nShield SCSI and PCI modules f/w 1.77.96   |
| 1.1.30  | 22 July 2002      | nForce and nShield SCSI and PCI modules f/w 2.0.0   |
| 1.1.33  | 4 October 2002    | nForce and nShield SCSI and PCI modules f/w 2.1.12  |
| 1.2.39  | 23 June 2003      | nCipher PMC module f/w 2.1.32   |
| 1.3.3   | 3 July 2003       | nForce and nShield PCI 800 modules f/w 2.0.1  |
| 1.3.6   | 5 September 2003  | nForce and nShield SCSI and PCI modules f/w 2.0.2   |
| 1.0.24  | 23 January 2004   | nForce and nShield SCSI f/w 2.0.5   |
| 1.3.14  | 18 March 2004     | nForce, nShield and Payshield SCSI and PCI modules f/w 2.12<br>adds nCipher 1600 PCI  |
| 1.4.20  | 5 October 2005    | nForce, nShield and Payshield SCSI and PCI modules f/w 2,18   |
| 2.0.0   | 6 April 2006      | nShield 500 PCI f/w 2.22.6  |
| 1.4.14  | 9 March 2006      | nForce and nShield SCSI f/w 1.77.100 and PCI modules f/w 2.12.9 and 2.18.15<br>Fix for security issues                          |
| 1.4.28  | 15 March 2006     | nForce and nShield SCSI f/w 1.77.100 and PCI modules f/w 2.12.9 and 2.18.15<br>Typographic corrections to above.                |
| 2.0.1   | 11 May 2006       | nShield 500, 2000 and 4000 PCI f/w 2.22.6<br>MiniHSM f/w 2.22.6   |

| Version | Date              | Comments   |
|---------|-------------------|--|
| 2.1.1   | 14 June 2006      | nShield 500, 2000 and 4000 PCI f/w 2.22.34                     |
| 2.1.2   | 29 August 2006    | MiniHSM build standard B                                       |
| 2.1.3   | 20 December 2006  | nShield 500 PCI f/w 2.22.34                                    |
| 2.2.2   | 29 April 2008     | nShield 500, 2000 and 4000 PCI f/w 2.2.43                      |
| 2.2.3   | 24 June 2008      | nShield 500 PCI and nShield 500, 2000 and 4000 PCI f/w 2.33.60 |
| 2.3.1   | 15 December 2008  | nShield PCI and nShield PCIe f/w 2.33,75                       |
| 2.4.1   | 28 August 2009    | nShield PCI and nShield PCIe f/w 2.33.82                       |
| 2.4.2   | 10 June 2009      | nShield PCI and nShield PCIe f/w 2.38.4                        |
| 2.5.3   | 28 January 2010   | nShield PCI and nShield PCIe f/w 2.33.82                       |
| 2.5.4   | 17 February 2010  | nShield PCI and nShield PCIe f/w 2.38.7                        |
| 3.0     | 11 May 2012       | nShield PCI and nShield PCIe f/w 2.50.17 - Thales branding     |
| 3.1     | 8 April 2014      | nShield Connect+ and nShield Solo+ range                       |
| 3.2     | 15 September 2015 | nShield PCI and nShield PCIe f/w 2.51.10, 2.50.35 and 2.55.1   |
| 3.3     | 03 November 2015  | nShield PCI and nShield PCIe f/w 2.51.10-2 and 2.55.1-2        |

---

# Contents

|   |           |
|---|-----------|
| <b>Versions</b> .....   | <b>3</b>  |
| <b>Contents</b> .....   | <b>5</b>  |
| <b>Chapter 1: Purpose</b> .....   | <b>6</b>  |
| <b>Chapter 2: Excluded Components</b> .....                                   | <b>9</b>  |
| <b>Chapter 3: Roles</b> .....   | <b>10</b> |
| <b>Chapter 4: Services available to each role</b> .....                       | <b>12</b> |
| <b>Chapter 5: Keys</b> .....  | <b>28</b> |
| <b>Chapter 6: Rules</b> .....   | <b>34</b> |
| 6.1 Identification and authentication .....                                   | 34        |
| 6.2 Procedures to initialise a module to comply with FIPS 140-2 Level 2 ..... | 36        |
| 6.3 Operating a level 2 module in FIPS mode .....                             | 37        |
| 6.4 To return a module to factory state .....                                 | 37        |
| 6.5 To create a new operator .....  | 38        |
| 6.6 To authorize the operator to create keys .....                            | 38        |
| 6.7 To authorize an operator to act as a Junior Security Officer .....        | 38        |
| 6.8 To authenticate an operator to use a stored key .....                     | 39        |
| 6.9 To authenticate an operator to create a new key .....                     | 39        |
| <b>Chapter 7: Physical security</b> .....                                     | <b>40</b> |
| <b>Chapter 8: Strength of functions</b> .....                                 | <b>41</b> |
| <b>Chapter 9: Self Tests</b> .....  | <b>44</b> |
| 9.1 Firmware Load Test .....  | 44        |
| <b>Chapter 10: Supported Algorithms</b> .....                                 | <b>45</b> |
| 10.1 FIPS approved and allowed algorithms: .....                              | 45        |
| 10.2 Non-FIPS approved algorithms .....                                       | 48        |
| <b>Addresses</b> .....  | <b>49</b> |

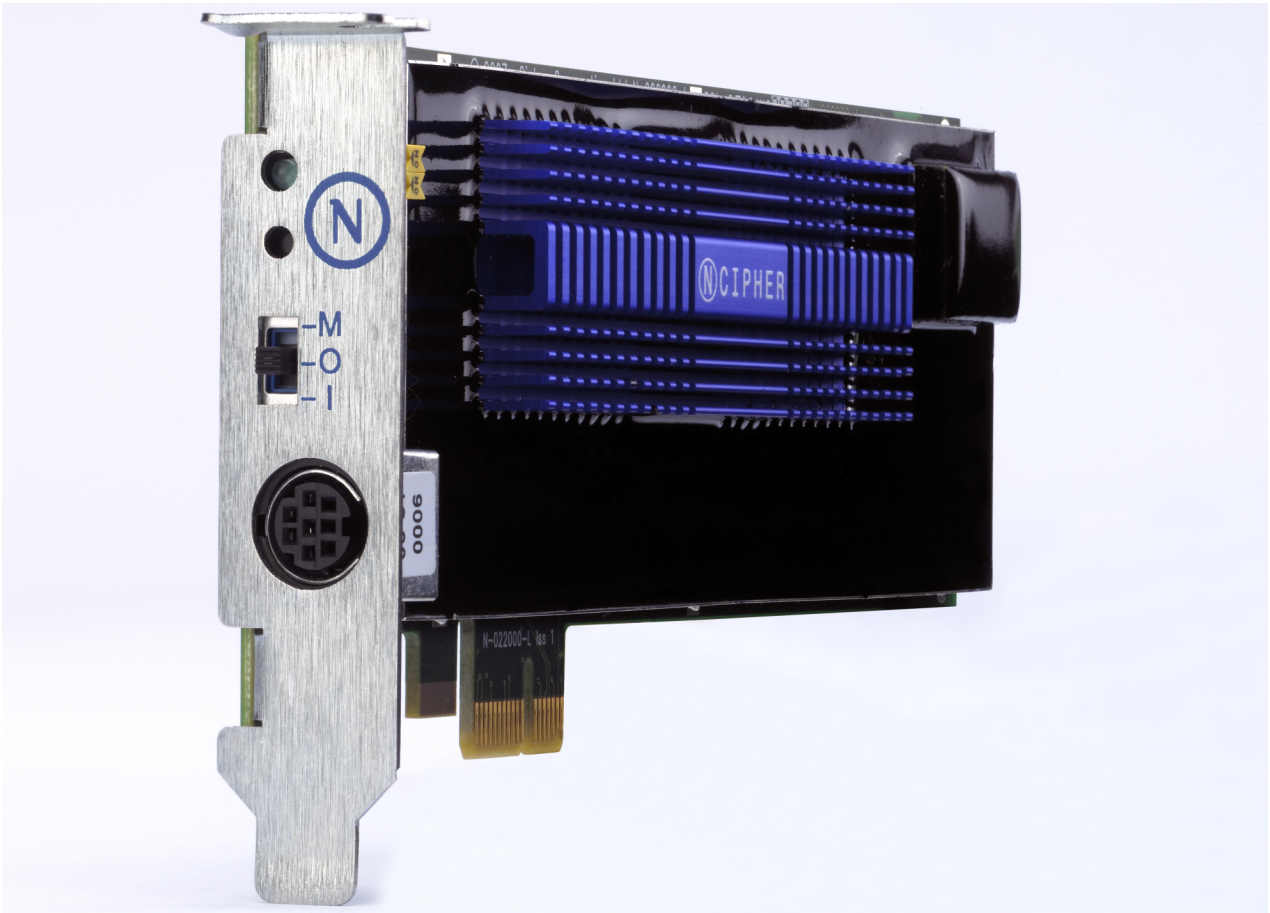
# Chapter 1: Purpose

nShield tamper evident and tamper responsive Hardware Security Modules provide support for the widest range of cryptographic algorithms, application programming interfaces (APIs) and host operating systems, enabling the devices to be used with virtually any business application—from identity management, web services and database encryption to tokenization, PKI services and strong authentication.

The nShield Hardware Security Modules are defined as multi-chip embedded cryptographic modules as defined by FIPS PUB 140-2.

| Unit ID                               | Model Number | Real Time Clock (RTC) NVRAM | Secure Execution Environment (SEE) | Potting (epoxy resin) | EMC classification | Crypto Accelerator | Overall FIPS level |
|---------------------------------------|--------------|-----------------------------|------------------------------------|-----------------------|--------------------|--------------------|--------------------|
| nShield F3 6000+                      | nC4433E-6K0  | Yes                         | Optional                           | Yes                   | B                  | Yes                | 2                  |
| nShield F3 500+                       | nC4433E-500  | Yes                         | Optional                           | Yes                   | B                  | Yes                | 2                  |
| nShield F3 10+                        | nC4033E-010  | Yes                         | Optional                           | Yes                   | B                  | None               | 2                  |
| nShield F3 6000+ for nShield Connect+ | nC4433E-6K0N | Yes                         | Optional                           | Yes                   | B                  | Yes                | 2                  |
| nShield F3 1500+ for nShield Connect+ | nC4433E-1K5N | Yes                         | Optional                           | Yes                   | B                  | Yes                | 2                  |
| nShield F3 500+ for nShield Connect+  | nC4433E-500N | Yes                         | Optional                           | Yes                   | B                  | Yes                | 2                  |

The units are identical in appearance and operation and only vary in the processing speed and the support software supplied.



All modules are supplied at build standard "N" or later to indicate that they meet the latest EU regulations regarding ROHS.

The nShield F3 6000+ is fitted in the Thales nShield Connect+, shown below:



Thales also supply modules to third party OEM vendors for use in a range of security products.

The module runs firmware provided by Thales. There is the facility for the administrator to upgrade this firmware. In order to determine that the module is running the correct version of firmware they should use the **NewEnquiry** service which reports the version of firmware currently loaded. The validated firmware version is 2.51.10-2 and 2.55.1-2.

The initialization parameters are reported by the **NewEnquiry** and **SignModuleState** services. An operator can determine which mode the module is operating in using the KeySafe GUI or the command line utilities supplied with the module, or their own code - these operate outside the security boundary.

The modules must be accessed by a custom written application. Full documentation for the nCore API can be downloaded from the Thales web site.

The modules have on-board non-volatile memory. There are services that enable memory to be allocated as files. Files have Access Control Lists that determine what operations can be performed on their contents. nShield modules have an on-board Real-time clock.

The module can be connected to a computer running one of the following operating systems:

- Windows
- Solaris
- HP-UX
- AIX
- Linux x86 / x64

Windows XP was used to test the module for this validation.

| Section                                      | Level    |
|--|----------|
| 1. Cryptographic Module Specification        | 2        |
| 2. Cryptographic Module Ports and Interfaces | 2        |
| 3. Roles, Services, and Authentication       | 3        |
| 4. Finite State Model                        | 2        |
| 5. Physical Security                         | 3        |
| 6. Operational Environment                   | N/A      |
| 7. Cryptographic Key Management              | 2        |
| 8. EMI/EMC                                   | 3        |
| 9. Self-Tests                                | 2        |
| 10. Design Assurance                         | 3        |
| 11. Mitigation of Other Attacks              | N/A      |
| <b>Overall FIPS Level</b>                    | <b>2</b> |



# Chapter 2: Excluded Components

The following components are excluded from FIPS 140-2 validation:

- Single lane PCIe bus
- Clear button
- Status LED
- Heat sink
- PS-2 Serial connector
- Mode switch
- 16-way header
- DIP switches
- SEE machine

# Chapter 3: Roles

The module defines the following roles: Unauthenticated, User, nShield Security Officer and Junior Security Officer. The nShield Security Officer and Junior Security Officer roles are equivalent of FIPS 140-2 Crypto-Officer role.

## Unauthenticated

All connections are initially unauthenticated.

A user in the unauthenticated role does not have access to handles or tickets required to provide access to the CSPs of authenticated users.

## User

An operator enters the user role by providing the required authority to carry out a service. The exact accreditation required to perform each service is listed in the table of services.

In order to perform an operation on a stored key, the operator must first load the key blob. If the key blob is protected by a logical token, the operator must first load the logical token by loading shares from smart cards.

Once an operator in the user role has loaded a key they can then use this key to perform cryptographic operations as defined by the Access Control List (ACL) stored with the key.

Each key blob contains an ACL that determines what services can be performed on that key. This ACL can require a certificate from an nShield Security Officer authorizing the action. Some actions including writing tokens always require a certificate.

## nShield Security Officer

The nShield Security Officer (NSO) is responsible for overall security of the module.

The nShield Security Officer is identified by a key pair, referred to as  $K_{NSO}$ . The hash of the public half of this key is stored when the unit is initialized. Any operation involving a module key or writing a token requires a certificate signed by  $K_{NSO}$ .

The nShield Security Officer is responsible for creating the authentication tokens (smart cards) for each operator and ensuring that these tokens are physically handed to the correct person.

An operator assumes the role of NSO by loading the private half of  $K_{NSO}$  and presenting the **ObjectID** for this key to authorize a command.

## Junior Security Officer

Where the nShield Security Officer want to delegate responsibility for authorizing an action they can create a key pair and give this to their delegate who becomes a Junior Security Officer (JSO). An ACL can then refer to this key, and the JSO is then empowered to sign the certificate authorizing the action. The JSO's keys should be stored on a key blob protected by a token that is not used for any other purpose.

In order to assume the role of JSO, the operator loads the JSO key and presents the **ObjectID** of this key, and if required the certificate signed by  $K_{NSO}$  that delegates authority to the key, to authorize a command.

A JSO can delegate portions of their authority to a new operator in the same way. The new operator will be a JSO if they have authority they can delegate, otherwise they will assume the user role.

# Chapter 4: Services available to each role

For more information on each of these services refer to the nCore Developer Tutorial and CodeSafe Developer Guide.

The following services provide authentication or cryptographic functionality. The functions available depend on whether the operator is in the unauthenticated role, the user or junior security officer (JSO) roles, or the nShield Security Officer (NSO) role. For each operation it lists the supported algorithms. Algorithms in square brackets are not under the operator's control. Algorithms used in optional portions of a service are listed in italics.

**Note:** Algorithms marked with an asterisk are not approved by NIST. If you choose to use them, the module is not operating in FIPS approved mode.

| Key Access | Description  |
|------------|--|
| Create     | Creates a in-memory object, but does not reveal value.                                   |
| Erase      | Erases the object from memory, smart card or non-volatile memory without revealing value |
| Export     | Discloses a value, but does not allow value to be changed.                               |
| Report     | Returns status information   |
| Set        | Changes a CSP to a given value   |
| Use        | Performs an operation with an existing CSP - without revealing or changing the CSP       |

| Command / Service | Role        |             |             | Description  | Key/CSP access   | Key types   |
|-------------------|-------------|-------------|-------------|--|--|---|
|                   | Unauth      | JSO / User  | NSO         |  |  |   |
| Bignum Operation  | Yes         | Yes         | Yes         | Performs simple mathematical operations.   | No access to keys or CSPs  |   |
| Change Share PIN  | No          | pass phrase | pass phrase | Updates the pass phrase used to encrypt a token share. The pass phrase supplied by the operator is not used directly, it is first hashed and then combined with the module key. To achieve this the command decrypts the existing share using the old share key derived from old pass phrase, module key and smart card identity. It then derives a new share key based on new pass phrase, module key and smart card identity, erases old share from smart card and writes a new share encrypted under the new share key. | Sets the pass phrase for a share, uses module key, uses share key, uses module key, creates share key, uses new share key, exports encrypted share, erases old share | [SHA-1 and AES or Triple DES]   |
| Channel Open      | handle, ACL | handle, ACL | handle, ACL | Opens a communication channel which can be used for bulk encryption or decryption.   | Uses a key object  | AES, Triple DES <sup>*</sup> , DES <sup>*</sup> , Arc Four <sup>*</sup> , Aria <sup>*</sup> , Camellia <sup>*</sup> , SEED <sup>*</sup> |
| Channel Update    | handle      | handle      | handle      | Performs encryption / decryption on a previously opened channel. The operation and key are specified in <b>ChannelOpen</b> .   | Uses a key object  | AES, Triple DES <sup>*</sup> , DES <sup>*</sup> , Arc Four <sup>*</sup> , Aria <sup>*</sup> , Camellia <sup>*</sup> , SEED <sup>*</sup> |
| Check User ACL    | handle      | handle      | handle      | Determines whether the ACL associated with a key object allows a specific operator defined action.   | Uses a key object  |   |
| Clear Unit        | Yes         | Yes         | Yes         | Zeroizes all loaded keys, tokens and shares. Clear Unit does not erase long term keys, such as module keys.  | Zeroizes objects.  | All   |

| Command / Service | Role        |             |             | Description  | Key/CSP access  | Key types  |
|-------------------|-------------|-------------|-------------|--|---|--|
|                   | Unauth      | JSO / User  | NSO         |  |   |  |
| Create Buffer     | No          | cert        | Yes         | Allocates an area of memory to load data. If the data is encrypted, this service specifies the encryption key and IV used. The decrypt operation is performed by <b>LoadBuffer</b>   | Uses a key object   | AES, Triple DES <sup>*</sup> , DES <sup>*</sup> , Arc Four <sup>*</sup> , Aria <sup>*</sup> , Camellia <sup>*</sup> , SEED <sup>*</sup>  |
| Decrypt           | handle, ACL | handle, ACL | handle, ACL | Decrypts a cipher text with a stored key returning the plain text.   | Uses a key object   | AES, Triple DES <sup>*</sup> , DES <sup>*</sup> , Arc Four <sup>*</sup> , Aria <sup>*</sup> , Camellia <sup>*</sup> , SEED <sup>*</sup> , RSA <sup>*</sup> , ElGamal <sup>*</sup> , KCDSA <sup>*</sup>   |
| Derive Key        | handle, ACL | handle, ACL | handle, ACL | The <b>DeriveKey</b> service provides functions that the FIPS 140-2 standard describes as key wrapping and split knowledge - it does not provide key derivation in the sense understood by FIPS 140-2. Creates a new key object from a variable number of other keys already stored on the module and returns a handle for the new key. This service can be used to split, or combine, encryption keys. This service is used to wrap keys according to the KDP so that a key server can distribute the wrapped key to micro-HSM devices. | Uses a key object, create a new key object.                                 | AES, AES key wrap, RSA, EC-DH, EC-MQV, Triple DES, PKCS #8 <sup>*</sup> , TLS key derivation, XOR, DLIES (D/H plus Triple DES or D/H plus AES) <sup>*</sup> , Aria <sup>*</sup> , Arc Four <sup>*</sup> , Camellia <sup>*</sup> , DES <sup>*</sup> , SEED <sup>*</sup> |
| Destroy           | handle      | handle      | handle      | Removes an object, if an object has multiple handles as a result of <b>RedeemTicket</b> service, this removes the current handle.  | Erases a <b>SEWorld</b> , <b>impath</b> , logical token, or any key object. | All  |
| Duplicate         | handle, ACL | handle, ACL | handle, ACL | Creates a second instance of a key object with the same ACL and returns a handle to the new instance.  | Creates a new key object.   | All  |

| Command / Service              | Role        |             |             | Description  | Key/CSP access  | Key types   |
|--------------------------------|-------------|-------------|-------------|--|---|---|
|                                | Unauth      | JSO / User  | NSO         |  |   |   |
| Encrypt                        | handle, ACL | handle, ACL | handle, ACL | Encrypts a plain text with a stored key returning the cipher text.   | Uses a key object                                     | AES, Triple DES, RSA*, DES*, ElGamal*, Arc Four*, Aria*, Camellia*, SEED*, KCDSA* |
| Erase File                     | Yes         | Yes         | Yes         | Removes a file, but not a logical token, from a smart card or software token.  | No access to keys or CSPs                             |   |
| Erase Share                    | Yes         | Yes         | Yes         | Removes a share from a smart card or software token.   | Erases a share  |   |
| Existing Client                | Yes         | Yes         | Yes         | Starts a new connection as an existing client.   | No access to keys or CSPs                             |   |
| Export                         | handle, ACL | handle, ACL | handle, ACL | If the unit is operating in FIPS level 2 mode this operation is only available for public keys - see <a href="#">Operating a level 2 module in FIPS mode on page 37</a> .  | Exports a [public] key object.                        | Any key type  |
| Feature Enable                 | No          | cert        | cert        | Enables a service. This requires a certificate signed by the Master Feature Enable key.  | Uses the public half of the Master Feature Enable Key | [DSA]   |
| Firmware Authenticate          | Yes         | Yes         | Yes         | Reports firmware version. Performs a zero knowledge challenge response protocol based on HMAC that enables a operator to ensure that the firmware in the module matches the firmware supplied by Thales. The protocol generates a random value to use as the HMAC key. | No access to keys or CSPs                             | HMAC  |
| Foreign Token Command (Bypass) | No          | handle      | handle      | Sends an ISO-7816 command to a smart card over the channel opened by <b>ForeignTokenOpen</b> .   | Uses a bypass channel.                                |   |

| Command / Service           | Role   |            |     | Description   | Key/CSP access  | Key types  |
|-----------------------------|--------|------------|-----|---|---|--|
|                             | Unauth | JSO / User | NSO |   |   |  |
| Foreign Token Open (Bypass) | No     | FE, cert   | FE  | Opens a channel to foreign smart card that accepts ISO-7816 commands. This service cannot be used if the smart card has been formatted using <b>FormatToken</b> . The channel is closed when the card is removed from the reader, or if the handle is destroyed. This service is feature enabled.   | Creates a bypass channel.   |  |
| FormatToken                 | Yes    | Yes        | Yes | Formats a smart card or software token ready for use.   | May use a module key to create challenge response value   | [AES, Triple DES]  |
| Generate Key                | Yes    | Yes        | Yes | Generates a symmetric key of a given type with a specified ACL and returns a handle. Optionally returns a certificate containing the ACL. The data generated by this operation is not a CSP until it has been bound to an authorized user by protecting it with a token.  | Creates a new symmetric key object. Sets the ACL and Application data for that object. Optionally uses module signing key and exports the key generation certificate. | AES, Triple DES, Arc Four*, Aria*, Camellia*, DES*, SEED.                      |
| Generate Key Pair           | Yes    | Yes        | Yes | Generates a key pair of a given type with specified ACLs for each half or the pair. Performs a pair wise consistency check on the key pair. Returns two key handles. Optionally returns certificates containing the ACL. The data generated by this operation is not a CSP until it has been bound to an authorized user by protecting it with a token. | Creates two new key objects. Sets the ACL and Application data for those objects. Optionally uses module signing key and exports two key generation certificates.     | Diffie-Hellman, DSA, DSA2, ECDSA, ECDSA2, EC-DH, EC-MQV, RSA, ElGamal*, KCDSA* |



| Command / Service               | Role        |             |             | Description   | Key/CSP access   | Key types           |
|---------------------------------|-------------|-------------|-------------|---|--|---------------------|
|                                 | Unauth      | JSO / User  | NSO         |   |  |                     |
| Generate KLF                    | No          | FE          | FE          | Generates a new long term key.<br>Never enabled in production units.  | <i>Erases</i> the module long term signing key, <i>creates</i> new module long term signing key. | [DSA and ECDSA]     |
| Generate Logical Token          | Yes         | Yes         | Yes         | Creates a new logical token, which can then be written as shares to smart cards or software tokens. On creation the token is not a CSP as it does not protect any sensitive data. | <i>Uses</i> module key. <i>Creates</i> a logical token.  | [AES or Triple DES] |
| Get ACL                         | handle, ACL | handle, ACL | handle, ACL | Returns the ACL for a given handle.   | <i>Exports</i> the ACL for a key object.   |                     |
| Get Application Data            | handle, ACL | handle, ACL | handle, ACL | Returns the application information stored with a key.  | <i>Exports</i> the application data of a key object.   |                     |
| Get Challenge                   | Yes         | Yes         | Yes         | Returns a random nonce that can be used in certificates   | No access to keys or CSPs  |                     |
| Get Key Info                    | handle      | handle      | handle      | Superseded by <b>GetKeyInfoExtended</b> , retained for compatibility.   | <i>Exports</i> the SHA-1 hash of a key object  |                     |
| Get Key Info Extended           | handle      | handle      | handle      | Returns the hash of a key for use in ACLs   | <i>Exports</i> the SHA-1 hash of a key object  |                     |
| Get Logical Token Info          | No          | handle      | handle      | Superseded by <b>GetLogicalTokenInfoExtended</b> , retained for compatibility.  | <i>Exports</i> the SHA-1 hash of a logical token.  | [SHA-1]             |
| Get Logical Token Info Extended | No          | handle      | handle      | Returns the token hash and number of shares for a logical token   | <i>Exports</i> the SHA-1 hash of a logical token.  | [SHA-1]             |
| Get Module Keys                 | Yes         | Yes         | Yes         | Returns a hashes of the nShield Security Officer's key and all loaded module keys.  | <i>Exports</i> the SHA-1 hash of KNSO and module keys.   | [SHA-1]             |

| Command / Service        | Role   |            |        | Description   | Key/CSP access   | Key types         |
|--------------------------|--------|------------|--------|---|--|-------------------|
|                          | Unauth | JSO / User | NSO    |   |  |                   |
| Get Module Long Term Key | Yes    | Yes        | Yes    | Returns a handle to the public half of the module's signing key. this can be used to verify key generation certificates and to authenticate inter module paths.   | Exports the public half of the module's long term signing key.           | [DSA, ECDSA]      |
| Get Module Signing Key   | Yes    | Yes        | Yes    | Returns the public half of the module's signing key. This can be used to verify certificates signed with this key.  | Exports the public half of the module's signing key.                     | [DSA2]            |
| Get RTC                  | Yes    | Yes        | Yes    | Reports the time according to the on-board real-time clock  | No access to keys or CSPs  |                   |
| Get Share ACL            | Yes    | Yes        | Yes    | Returns the access control list for a share   | Exports the ACL for a token share on a smart card.                       |                   |
| Get Slot Info            | Yes    | Yes        | Yes    | Reports status of the physical token in a slot. Enables an operator to determine if the correct token is present before issuing a <b>ReadShare</b> command. If the token was formatted with a challenge response value, uses the module key to authenticate the smart card. | Uses a module key if token is formatted with a challenge response value. | [AES, Triple DES] |
| Get Slot List            | Yes    | Yes        | Yes    | Reports the list of slots available from this module.   | No access to keys or CSPs  |                   |
| GetTicket                | handle | handle     | handle | Gets a ticket - an invariant identifier - for a key. This can be passed to another client or to a <b>SEE World</b> which can redeem it using <b>RedeemTicket</b> to obtain a new handle to the object,  | Uses a key object, logical token, impath, <b>SEWorld</b> .               |                   |
| Get World Signers        | No     | handle     | handle | Returns a list of the keys used to sign a <b>SEWorld</b> identified by the key hash and the signing mechanism used. This command can only be called from inside the SEE world.  | Uses a <b>SEWorld</b> object.  |                   |

| Command / Service          | Role   |            |        | Description  | Key/CSP access  | Key types   |
|----------------------------|--------|------------|--------|--|---|---|
|                            | Unauth | JSO / User | NSO    |  |   |   |
| Hash                       | Yes    | Yes        | Yes    | Hashes a value.  | No access to keys or CSPs   | SHA-1, SHA-256, SHA-384, SHA-512                  |
| Impath Get Info            | No     | handle     | handle | Reports status information about an impath   | Uses an Impath, exports status information.   |   |
| Impath Key Exchange Begin  | FE     | FE         | FE     | Creates a new inter-module path and returns the key exchange parameters to send to the peer module.  | Creates a set of Impath keys  | [DSA2 or DSA and Diffie Hellman] AES, Triple-DES  |
| Impath Key Exchange Finish | No     | handle     | handle | Completes an impath key exchange. Require the key exchange parameters from the remote module.  | Creates a set of Impath keys.   | [DSA2 or DSA and Diffie Hellman, AES, Triple DES] |
| Impath Receive             | No     | handle     | handle | Decrypts data with the Impath decryption key.  | Uses an Impath key.   | [AES or Triple DES]                               |
| Impath Send                | No     | handle     | handle | Encrypts data with the impath encryption key.  | Uses an Impath key.   | [AES or Triple DES]                               |
| Import                     | Yes    | Yes        | Yes    | Loads a key and ACL from the host and returns a handle.<br>The data generated by this operation is not a CSP until it has been bound to an authorized user by protecting it with a token.<br>If the unit is operating in FIPS mode at level 2, this operation must only be used for public keys - see <a href="#">Operating a level 2 module in FIPS mode on page 37</a> | Creates a new key object to store imported key, sets the key value, ACL and App data. |   |

| Command / Service | Role   |            |        | Description  | Key/CSP access   | Key types   |
|-------------------|--------|------------|--------|--|--|---|
|                   | Unauth | JSO / User | NSO    |  |  |   |
| Initialise        | init   | init       | init   | <p>Initializes the module, returning it to the factory state. This clears all NVRAM files, all loaded keys and all module keys and the module signing key. This can only be performed when the module is in initialisation mode.</p> <p>It also generates a new KM0 and module signing key. The only key that is not zeroized is the long term signing key. This key only serves to provide a cryptographic identity for a module that can be included in a PKI certificate chain. Thales may issue such certificates to indicate that a module is a genuine nShield module. This key is not used to encrypt any other data.</p> | <i>Erases all keys, Creates KM0 and KML</i>  | [DSA2]  |
| Load Blob         | No     | handle     | handle | Loads a key that has been stored in a key blob. The operator must first have loaded the token or key used to encrypt the blob.   | <i>Uses module key, logical token, or archiving key, creates a new key object.</i> | Triple DES and SHA-1 or AES, DH, or RSA plus AES, SHA-1, and HMAC SHA-1 |
| Load Buffer       | No     | handle     | handle | Loads signed data into a buffer. Several load buffer commands may be required to load all the data, in which case it is the responsibility of the client program to ensure they are supplied in the correct order. Requires the handle of a buffer created by <b>CreateBuffer</b> .  | No access to keys or CSPs  |   |

| Command / Service  | Role   |             |             | Description  | Key/CSP access   | Key types   |
|--------------------|--------|-------------|-------------|--|--|---|
|                    | Unauth | JSO / User  | NSO         |  |  |   |
| Load Logical Token | yes    | yes         | yes         | Allocates space for a new logical token - the individual shares can then be assembled using <b>ReadShare</b> or <b>ReceiveShare</b> . Once assembled the token can be used in <b>LoadBlob</b> or <b>MakeBlob</b> commands. | Uses module key  | [AES or Triple DES]   |
| Make Blob          | No     | handle, ACL | handle, ACL | Creates a key blob containing the key and returns it. The key object to be exported may be any algorithm.  | Uses module key, logical token or archiving key, exports encrypted key object. | Triple DES and SHA-1 or AES, DH, or RSA plus AES, SHA-1, and HMAC SHA-256 or HMAC SHA-1 |
| Mod Exp            | Yes    | Yes         | Yes         | Performs a modular exponentiation on values supplied with the command.   | No access to keys or CSPs  |   |
| Mod Exp CRT        | Yes    | Yes         | Yes         | Performs a modular exponentiation on values, supplied with the command using Chinese Remainder Theorem.  | No access to keys or CSPs  |   |
| Module Info        | Yes    | Yes         | Yes         | Reports low level status information about the module. This service is designed for use in Thales' test routines.  | No access to keys or CSPs  |   |
| NewClient          | Yes    | Yes         | Yes         | Returns a client id.   | No access to keys or CSPs  |   |
| New Enquiry        | Yes    | Yes         | Yes         | Reports status information.  | No access to keys or CSPs  |   |
| No Operation       | Yes    | Yes         | Yes         | Does nothing, can be used to determine that the module is responding to commands.  | No access to keys or CSPs  |   |
| NVMem Allocate     | No     | cert        | Yes         | Allocates an area of non-volatile memory as a file and sets the ACLs for this file. This command can now be used to write files protected by an ACL to a smart card  | No access to keys or CSPs  |   |
| NVMem Free         | No     | cert        | Yes         | Frees a file stored in non-volatile memory. This command can now be used to write files protected by an ACL to a smart card  | No access to keys or CSPs  |   |

| Command / Service | Role   |            |     | Description   | Key/CSP access   | Key types |
|-------------------|--------|------------|-----|---|--|-----------|
|                   | Unauth | JSO / User | NSO |   |  |           |
| NVMem List        | Yes    | Yes        | Yes | Reports a list of files stored in the non-volatile memory or protected by an ACL on a smart card.   | No access to keys or CSPs  |           |
| NVMem Operation   | No     | cert, ACL  | ACL | Performs an operation on a file stored in non-volatile memory. Operations include: read, write, increment, decrement, etc. This command can now be used to write files protected by an ACL to a smart card  | No access to keys or CSPs  |           |
| Random Number     | Yes    | Yes        | Yes | Generates a random number for use in a application using the on-board random number generator. There are separate services for generating keys. The random number services are designed to enable an application to access the random number source for its own purposes - for example an on-line casino may use <b>GenerateRandom</b> to drive its applications. | Uses DRBG key  | [AES]     |
| Random Prime      | Yes    | Yes        | Yes | Generates a random prime. This uses the same mechanism as is used for RSA and Diffie-Hellman key generation. The primality checking conforms to ANSI X9.31.   | Uses DRBG key  | [AES]     |
| Read File         | Yes    | Yes        | Yes | Reads a file, but not a logical token, from a smart card or software token. This command can only read files without ACLs.  | Reads a file, but not a logical token, from a smart card or software token. This command can only read files without ACLs. No access to keys or CSPs |           |

| Command / Service    | Role        |                         |                         | Description   | Key/CSP access   | Key types                  |
|----------------------|-------------|-------------------------|-------------------------|---|--|----------------------------|
|                      | Unauth      | JSO / User              | NSO                     |   |  |                            |
| Read Share           | Yes         | Yes                     | Yes                     | Reads a share from a physical token.<br><br>Once sufficient shares have been loaded recreates token-may require several <b>ReadShare</b> or <b>ReceiveShare</b> commands.           | Uses pass phrase, module key, creates share key, uses share key, creates a logical token.                    | [SHA-1, AES or Triple DES] |
| Receive Share        | No          | handle, encrypted share | handle, encrypted share | Takes a share encrypted with <b>SendShare</b> and a pass phrase and uses them to recreate the logical token. - may require several <b>ReadShare</b> or <b>ReceiveShare</b> commands | Uses an Impath key, uses pass phrase, module key, creates share key, uses share key, creates a logical token | [AES, Triple DES]          |
| Redeem Ticket        | ticket      | ticket                  | ticket                  | Gets a handle in the current name space for the object referred to by a ticket created by <b>GetTicket</b> .  | Uses a key object, logical token, impath, or <b>SEWorld</b> .  |                            |
| Remove KM            | No          | cert                    | Yes                     | Removes a loaded module key.  | Erases a module key  |                            |
| Send Share           | No          | handle, ACL             | handle, ACL             | Reads a logical token share and encrypts it under an impath key for transfer to another module where it can be loaded using <b>ReceiveShare</b>                                     | Uses an impath key, exports encrypted share.   | [AES, Triple DES]          |
| Set ACL              | handle, ACL | handle, ACL             | handle, ACL             | Sets the ACL for an existing key. The existing ACL for the key must allow the operation.  | Sets the Access Control List for a key object  |                            |
| Set Application Data | handle, ACL | handle, ACL             | handle, ACL             | Stores information with a key.  | Sets the application data stored with a key object   |                            |
| Set KM               | No          | cert                    | Yes                     | Loads a key object as a module key.   | Uses a key object, sets a module key   | AES, Triple DES            |

| Command / Service         | Role        |             |             | Description  | Key/CSP access   | Key types   |
|---------------------------|-------------|-------------|-------------|--|--|---|
|                           | Unauth      | JSO / User  | NSO         |  |  |   |
| Set NSO Perm              | init        | init        | No          | Designates a key hash as the nShield Security Officer's Key and sets the security policy to be followed by the module. This can only be performed while the unit is in the initialisation state. | Sets the identity of the nShield Security officer's key. | [SHA-1 hash of DSA key]                                     |
| Set RTC                   | No          | cert        | Yes         | Sets the real-time clock.  | No access to keys or CSPs                                |   |
| Sign                      | handle, ACL | handle, ACL | handle, ACL | Returns the digital signature or MAC of plain text using a stored key.   | Uses a key object  | RSA, DSA, DSA2, ECDSA, ECDSA2, Triple DES MAC, HMAC, *KCDSA |
| Sign Module State         | handle, ACL | handle, ACL | handle, ACL | Signs a certificate describing the modules security policy, as set by SetNSOPerm.  | Uses the module signing key                              | [DSA]   |
| Statistics Enumerate Tree | Yes         | Yes         | Yes         | Reports the statistics available.  | No access to keys or CSPs                                |   |
| Statistic Get Value       | Yes         | Yes         | Yes         | Reports a particular statistic.  | No access to keys or CSPs                                |   |



| Command / Service  | Role        |             |             | Description   | Key/CSP access   | Key types   |
|--|-------------|-------------|-------------|---|--|---|
|  | Unauth      | JSO / User  | NSO         |   |  |   |
| Update Firmware Service<br>(Calls: Programming Begin, Programming Begin Chunk, Programming Load Block, Programming End Chunk, Programming End) | monitor     | monitor     | monitor     | <p>These commands are used in the update firmware service. The individual commands are required to load the candidate firmware image in sections small enough to be transported by the interface. Thales supply the <b>LoadROM</b> utility for the administrator to use for this service. This utility issues the correct command sequence to load the new firmware. The module will only be operating in a FIPS approved mode if you install firmware that has been validated by NIST / CSEC. Administrators who require FIPS validation should only upgrade firmware after NIST / CSEC issue a new certificate. The monitor also checks that the Version Sequence Number (VSN) of the firmware is as high or higher than the VSN of the firmware currently installed.</p> | <p>Uses Firmware Integrity Key and Firmware Confidentiality Keys.<br/>Sets Firmware Integrity Key and Firmware Confidentiality Keys.</p> | [DSA2, AES]   |
| Verify   | handle, ACL | handle, ACL | handle, ACL | Verifies a digital signature using a stored key.  | Uses a key object.   | RSA, DSA, DSA2, ECDSA, ECDSA2, Triple DES MAC, HMAC, *KCDSA |
| Write File   | Yes         | Yes         | Yes         | Writes a file, but not a logical token, to a smart card or software token. Note these files do not have an ACL, use the NVMEM commands to create files with an ACL.   | No access to keys or CSPs  |   |

| Command / Service | Role   |            |        | Description  | Key/CSP access   | Key types                |
|-------------------|--------|------------|--------|--|--|--------------------------|
|                   | Unauth | JSO / User | NSO    |  |  |                          |
| Write Share       | No     | handle     | handle | Writes a new share to a smart card or software token. The number of shares that can be created is specified when the token is created. All shares must be written before the token is destroyed. | <i>Sets pass phrase, uses module key, creates share, uses pass phrase and module key, creates share key, uses module key, uses share key, exports encrypted share.</i> | [AES, Triple DES, SHA-1] |

## Terminology

| Code            | Description  |
|-----------------|--|
| No              | The operator can not perform this service in this role.  |
| yes             | The operator can perform this service in this role without further authorization.  |
| handle          | <p>The operator can perform this service if they possess a valid handle for the resource: key, channel, impath, token, buffers.</p> <p>The handle is an arbitrary number generated when the object is created.</p> <p>The handle for an object is specific to the operator that created the object.</p> <p>The ticket services enable an operator to pass an ID for an object they have created to another operator.</p> |
| ACL             | <p>Access Control List. The operator can only perform this service with a key if the ACL for the key permits this service. The ACL may require that the operator present a certificate signed by a Security Officer or another key.</p> <p>The ACL may specify that a certificate is required, in which case the module verifies the signature on the certificate before permitting the operation.</p>                   |
| pass phrase     | An operator can only load a share, or change the share PIN, if they possess the pass phrase used to derive the share. The module key with which the pass phrase was combined must also be present.   |
| cert            | An operator can only perform this service if they are in possession of a certificate from the nShield Security Officer. This certificate will reference a key. The module verifies the signature on the certificate before permitting the operation.   |
| FE              | Feature Enable. This service is not available on all modules. It must be enabled using the <b>FeatureEnable</b> service before it can be used.   |
| encrypted share | The <b>ReceiveShare</b> command requires a logical token share encrypted using an Impath key created by the <b>SendShare</b> command.  |
| ticket          | The <b>RedeemTicket</b> command requires the ticket generated by <b>GetTicket</b> .  |
| init            | These services are used to initialise the module. They are only available when the module is in the initialisation mode. To put the module into initialisation mode you must have physical access to the module and put the mode switch into the initialisation setting. In order to restore the module to operational mode you must put the mode back to the Operational setting.                                       |
| monitor         | These services are used to reprogram the module. They are only available when the module is in the monitor mode. To put the module into monitor mode you must have physical access to the module and put the mode switch into the monitor setting. In order to restore the module to operational mode you reinitialize the module and then return it to operational state.   |

# Chapter 5: Keys

For each type of key used by the nShield modules, the following section describes the access that a operator has to the keys.

nShield modules refer to keys by their handle, an arbitrary number, or by its SHA-1 hash.

## nShield Security Officer's key

The nShield Security officer's key must be set as part of the initialisation process. This is a public / private key pair that the nShield Security Officer uses to sign certificates to authorize key management and other secure operations.

The SHA-1 hash of the public half of this key pair is stored in the module EEPROM .

The public half of this key is included as plain text in certificates.

The module treats anyone in possession of the private half of this key as the nShield Security Officer.

If you use the standard tools supplied by Thales to initialise the module, then this key is a DSA key stored as a key blob protected by a logical token on the Administrator Card Set.

## Junior Security Officer's key

Because the nShield Security Officer's key has several properties, it is good practice to delegate authority to one or more Junior Security Officers, each with authority for defined operations.

To create a Junior Security Officer (JSO) the NSO creates a certificate signing key for use as their JSO key. This key must be protected by a logical token in the same manner as any other application key.

Then to delegate authority to the JSO, the nShield Security Officer creates a certificate containing an Access Control List specifying the authority to be delegated and the hash of the JSO key to which the powers are to be delegated.

The JSO can then authorize the actions listed in the ACL - as if they were the NSO - by presenting the JSO key and the certificate. If the JSO key is created with the Sign permission in its ACL, the JSO may delegate parts of their authority to another key. The holder of the delegate key will need to present the certificate signed by the NSO and the certificate signed by the JSO. If the JSO key only has **UseAsCertificate** permissions, then they cannot delegate authority.

If you use the standard tools supplied by Thales to initialise the module, then this key is a DSA key stored as a key blob protected by a logical token on the Administrator Card Set.

## Long term signing key

The nShield modules store one 160-bit and one 256-bit random number in the EEPROM .

The 160-bit number is combined with a discrete log group stored in the module firmware to produce a DSA key. The 256-bit number is used as the private exponent of a ECDSA key using the NIST P521 curve.

It can be used to sign a module state certificate using the **SignModuleState** service and the public value retrieved by the non-cryptographic service **GetLongTermKey**.

This is the only key that is not zeroized when the module is initialized.

This key is not used to encrypt any other data. It only serves to provide a cryptographic identity for a module that can be included in a PKI certificate chain. Thales may issue such certificates to indicate that a module is a genuine Thales module.

## Module signing key

When the nShield module is initialized it automatically generates a 3072-bit DSA2 key pair that it uses to sign certificates. Signatures with this key use SHA-256. The private half of this pair is stored internally in EEPROM and never released. The public half is revealed in plaintext, or encrypted as a key blob. This key is only ever used to verify that a certificate was generated by a specified module.

## Module keys

Module keys are AES or Triple DES used to protect tokens. The nShield modules generates the first module key  $K_{M0}$  when it is initialized. This module key is guaranteed never to have been known outside this module.  $K_{M0}$  is an AES key. The nShield Security Officer can load further module keys. These can be generated by the module or may be loaded from an external source. Setting a key as a module key stores the key in EEPROM .

Module keys can not be exported once they have been assigned as module keys. They may only be exported on a key blob when they are initially generated.

## Logical tokens

A logical token is an AES or Triple DES key used to protect key blobs. Logical tokens are associated with module keys. The key type depends on the key type of the module key.

When you create a logical token, you must specify parameters, including the total number of shares, and the number or shares required to recreate the token, the quorum. The total number can be any integer between 1 and 64 inclusive. The quorum can be any integer from 1 to the total number.

A logical token is always generated randomly, using the on-board random number generator.

While loaded in the module logical tokens are stored in the object store.

Token keys are never exported from the module, except on physical tokens or software tokens. When a logical token is exported the logical token - the key data plus the token parameters - is first encrypted with a module key. Then the encrypted token is split into shares using the Shamir Threshold Sharing algorithm, even if the total number of shares is one. Each share is then encrypted using a share key (which may require knowledge of a passphrase to derive - see Share keys) and written to a physical token - a smart card - or a software token. Logical tokens can be shared between one or more physical token. The properties for a token define how many shares are required to recreate the logical token. Shares can only be generated when a token is created. The firmware prevents a specific share from being written more than once.

Logical tokens are not used for key establishment.

## Share keys

A share key is used to protect a logical token share when they are written to a smart card or software token that is used for authentication. The share key is created by creating a message comprised of a secret prefix, Module key, Share number, smart card unique id and an optional 20 bytes supplied by the operator (expected to be the SHA-1 hash of a pass phrase entered into the application), and using this as the input to a PRNG to form a unique key used to encrypt the share - this is either an AES or Triple DES key depending on the key type of the logical token which is itself determined by the key type of the module key. This key is not stored on the module. It is recalculated every time share is loaded. The share data includes a MAC, if the MAC does not verify correctly the share is rejected.

The share key is not used directly to protect CSPs nor is the Share Key itself considered a CSP. It is used for authentication only. The logical token needs to be reassembled from the shares using Shamir Threshold Sharing Scheme and then be decrypted using the module key. Only then can the logical token be used to decrypt application keys.

## Impath keys

An impath is a secure channel between two modules.

To set up an impath two modules perform a key-exchange, using 3072-bit Diffie-Hellman.

The Diffie Hellman operations has been validated in CVL Cert. #1. The CVL Cert. #1 is not fully compliant to SP 800-56A as the key derivation function has not been tested.

The key exchange parameters for each module are signed by that module's signing key. Once the modules have validated the signatures the module derives four symmetric keys for cryptographic operations using a protocol based on SHA-1 or SHA-256.

Currently symmetric keys are AES or Triple DES. AES is used if both modules use 2.50.16 or later firmware, Triple DES is used where the other module is running older firmware. The four keys are used for encryption, decryption, MAC creation, MAC validation. The protocol ensures that the key Module 1 uses for encryption is used for decryption by module 2. All impath keys are stored as objects in the object store in SRAM.

## Key objects

Keys used for encryption, decryption, signature verification and digital signatures are stored in the module as objects in the object store in RAM. All key objects are identified by a random identifier that is specific to the operator and session.

All key objects are stored with an Access control List or ACL. The ACL specifies what operations can be performed with this key. Whenever an operator generates a key or imports a key in plain text they must specify a valid ACL for that key type. The ACL can be changed using the **SetACL** service. The ACL can only be made more permissive if the original ACL includes the **ExpandACL** permission.

Key objects may be exported as key blobs if their ACL permits this service. Each blob stores a single key and an ACL. The ACL specifies what operations can be performed with this copy of the key. The ACL stored with the blob must be at least as restrictive as the ACL associated with the key object from which the blob was created. When you load a key blob, the new key object takes its ACL from the key blob. Working key blobs are encrypted under a logical token. Key objects may also be exported as key blobs under an archiving key. The key blob can be stored on the host disk or in the module NVRAM.

Key objects can only be exported in plain text if their ACL permits this operation. An operator may pass a key to another operator - or to a SEE World - using the ticketing mechanism. The **GetTicket** mechanism takes a key identifier and returns a ticket. This ticket refers to the key identifier - it does not include any key data. A ticket can be passed as a byte block to the other operator who can then use the **RedeemTicket** service to obtain a key identifier for the same object that is valid for their session. As the new identifier refers to the same object the second operator is still bound by the original ACL.

## Session keys

Keys used for a single session are generated as required by the module. They are stored along with their ACL as objects in the object store. These may be of any supported algorithm.

## Archiving keys

It is sometimes necessary to create an archive copy of a key, protected by another key. Keys may be archived using:

- Three-key Triple DES keys (used for unwrapping legacy keys and wrapping public keys only).
- A combination of three-key Triple DES and RSA keys (unwrapping legacy keys only). In this case a random 168-bit Triple DES key is created which is used to encrypt working key and then this key is wrapped by the RSA key.
- A scheme using RSA. 3072-bit RSA is used to establish a secret from which encryption keys are generated. The holders of the public and private halves of the RSA key must already exist in the module as operators.

The keys generated are either AES or Triple-DES keys, for the purpose of protecting other keys. AES is used by default as of firmware version 2.50.35. (with Triple-DES available for legacy purposes).

Once the key agreement process is complete, the module uses an additional keyed hashing process to protect the integrity of the nCore Key object to be archived, which is comprised of the key type, key value and Access Control List. This process uses HMAC SHA-256 by default. (with HMAC SHA-1 available for legacy purposes).

- A scheme using Diffie Hellman:

3072-bit Diffie-Hellman, which is allowed for use in the Approved mode, is used to establish a secret from which encryption keys are generated. Both parties in the Diffie-Hellman key agreement process exist in the module as operators. The keys generated are either AES or Triple-DES keys, for the purpose of protecting other keys. AES is used by default as of firmware version 2.50.35. (with Triple-DES available for legacy purposes). Please note that the Diffie-Hellman private key must be stored externally on the smartcard, if the archived keys are to be retrieved at a later time.

Once the key agreement process is complete, the module uses an additional keyed hashing process to protect the integrity of the nCore Key object to be archived, which is comprised of the key type, key value and Access Control List. This process uses HMAC SHA-256 by default. (with HMAC SHA-1 available for legacy purposes).

Although provided by the firmware, this option is currently not used by any Thales tools.

Other third party applications external to the module, may take advantage of this option, at the discretion of the developer.

When a key is archived in this way it is stored with its ACL.

When you generate or import the archiving key, you must specify the **UseAsBlobKey** option in the ACL. The archiving key is treated as any other key object.

When you generate or import the key that you want to archive you must specify the Archival options in the ACL. This options can specify the hash(es) of the allowed archiving key(s). If you specify a list of hashes, no other key may be used.

## Certificate signing keys

The ACL associated with a key object can call for a certificate to be presented to authorize the action. The required key can either be the nShield Security Officer's key or any other key. Keys are specified in the ACL by an identifying key SHA-1 hash. The key type is also specified in the ACL although DSA is standard, any signing algorithm may be used, all Thales tools use DSA.

Certain services can require certificates signed by the nShield Security Officer.

## Firmware Integrity Key

All firmware is signed using a 3072-bit DSA2 key pair. Signatures with this key use SHA-256.

The module checks the signature before new firmware is written to flash. A module only installs new firmware if the signature decrypts and verifies correctly.

The private half of this key is stored at Thales.



The public half is included in all firmware. The firmware is stored in flash memory when the module is switched off, this is copied to RAM as part of the start up procedure.

## Firmware Confidentiality Key

All firmware is encrypted using AES to prevent casual decompilation.

The encryption key is stored at Thales' offices and is in the firmware.

The firmware is stored in flash memory when the module is switched off, this is copied to RAM as part of the start up procedure.

## Master Feature Enable Key

For commercial reasons not all devices in the nShield family of HSMs offer all services. Certain services must be enabled separately. In order to enable a service the operator presents a certificate signed by the Master Feature Enable Key.

The Master Feature Enable Key is a DSA key pair. The private half of this key pair is stored at Thales' offices. The public half of the key pair is included in the firmware. The firmware is stored in flash memory when the module is switched off, this is copied to RAM as part of the start up procedure.

## DRBG Key

DRBG stands for Deterministic Random Bit Generator.

The module uses the CTR\_DRBG from SP800-90 with a 256-bit AES key. This key is seeded from the on board entropy source whenever the module is initialised and is reseeded according to SP800-90 with a further 512 bits of entropy after every 2048-bytes of output.

This key is only ever used by the DRBG. It is never exposed outside the module.

The DRBG internal state is contained within the DRBG mechanism boundary and is not accessed by non-DRBG functions or by other instances of any DRBG.

**Note:** For CTR DRBG, the values of V and Key (SP 800-90) are the 'secret values' of the internal state.

# Chapter 6: Rules

## 6.1 Identification and authentication

Communication with the modules is performed via a server program running on the host machine, using standard inter process communication, using sockets in UNIX operating systems, named pipes under Windows.

In order to use the module the operator must first log on to the host computer and start an nShield enabled application. The application connects to the hardware server, this connection is given a client ID, a 32-bit arbitrary number.

Before performing any service the operator must present the correct authorization. Where several stages are required to assemble the authorization, all the steps must be performed on the same connection. The authorization required for each service is listed in the section [Services available to each role on page 12](#). An operator cannot access any service that accesses CSPs without first presenting a smart card, or software token.

The nShield modules performs identity based authentication. Each individual operator is given a smart card that holds their authentication data - the logical token share - in an encrypted form. All operations require the operator to first load the logical token from their smart card.

### Access Control

Keys are stored on the host computer's hard disk in an encrypted format, known as a key blob. In order to load a key the operator must first load the token used to encrypt this blob.

Tokens can be divided into shares. Each share can be stored on a smart card or software token on the computer's hard disk. These shares are further protected by encryption with a pass phrase and a module key. Therefore an operator can only load a key if they possess the physical smart cards containing sufficient shares in the token, the required pass phrases and the module key are loaded in the module.

Module keys are stored in EEPROM in the module. They can only be loaded or removed by the nShield Security Officer, who is identified by a public key pair created when the module is initialized. It is not possible to change the nShield Security Officer's key without re-initializing the module, which clears all the module keys, thus preventing access to all other keys.

The key blob also contains an Access Control List that specifies which services can be performed with this key, and the number of times these services can be performed. These can be hard limits or per authorization limits. If a hard limit is reached that service can no longer be performed on that key. If a per-authorization limit is reached the operator must reauthorize the key by reloading the token.

All objects are referred to by handle. Handles are cross-referenced to **ClientIDs**. If a command refers to a handle that was issued to a different client, the command is refused. Services exist to pass a handle between **ClientIDs**.

## Access Control List

All key objects have an Access Control List (ACL). The operator must specify the ACL when they generate or import the key. The ACL lists every operation that can be performed with the key - if the operation is not in the ACL the module will not permit that operation. In particular the ACL can only be altered if the ACL includes the **SetACL** service. The ACL is stored with the key when it is stored as a blob and applies to the new key object created when you reload the blob.

The ACL can specify limits on operations - or groups of operations - these may be global limits or per authorization limits. If a global limit is exceeded then the key cannot be used for that operation again. If a per authorization limit is exceeded then the logical token protecting the key must be reloaded before the key can be reused.

An ACL can also specify a certifier for an operation. In this case the operator must present a certificate signed by the key whose hash is in the ACL with the command in order to use the service.

An ACL can also list Operator Defined actions. These actions do not permit any operations within the module, but can be tested with the **CheckUserACL** service. This enables SEE programs to make use of the ACL system for their own purposes. For example payShield uses this feature to determine the role of a Triple-DES key within EMV.

An ACL can also specify a host service identifier. In which case the ACL is only met if the hardserver appends the matching Service name. This feature is designed to provide a limited level of assurance and relies on the integrity of the host, it offers no security if the server is compromised or not used.

ACL design is complex - operators will not normally need to write ACLs themselves. Thales provide tools to generate keys with strong ACLs.

## Object re-use

All objects stored in the module are referred to by a handle. The module's memory management functions ensure that a specific memory location can only be allocated to a single handle. The handle also identifies the object type, and all of the modules enforce strict type checking. When a handle is released the memory allocated to it is actively zeroed.

## Error conditions

If the module cannot complete a command due to a temporary condition, the module returns a command block with no data and with the status word set to the appropriate value. The operator can resubmit the command at a later time. The server program can record a log of all such failures.

If the module encounters an unrecoverable error it enters the error state. This is indicated by the status LED flashing in the Morse pattern SOS. As soon as the unit enters the error state all processors stop processing commands and no further replies are returned. In the error state the unit does not respond to commands. The unit must be reset.

## Security Boundary

The physical security boundary is the plastic jig that contains the potting on both sides of the PCB.

All cryptographic components are covered by potting.

There is also a logical security boundary between the nCore kernel and the SEE.

Some items are excluded from FIPS 140-2 validation as they are not security relevant see [Excluded Components on page 9](#).

## Status information

The module has a status LED that indicates the overall state of the module.

The module also returns a status message in the reply to every command. This indicates the status of that command.

There are a number of services that report status information.

Where the module is fitted inside an nShield Connect+ this information can be displayed on the LCD on the nShield Connect+'s front panel.

## 6.2 Procedures to initialise a module to comply with FIPS 140-2 Level 2

The nShield enabled application must perform the following services, for more information refer to the nShield User Guide.

Put the mode switch into the initialisation position and restart the module

Use either the graphical user interface KeySafe or the command line tool **new-world**. Using either tool you must specify the number of cards in the Administrator Card Set and the encryption algorithm to use, Triple-DES or AES. To ensure that the module is in Level 2 mode you must:

Using Keysafe select the option "**FIPS 140 Mode level 3 compliant**" = **No**

Using **new-world** do NOT specify the **-F** flag in the command line

The tool prompts you to insert cards and to enter a pass phrase for each card.

When you have created all the cards, reset the mode switch into the operational position and restart the module.

## Verifying the module is in level 2 mode

An operator can verify the initialisation status of the module as if a module is initialised in level 2 mode

Keysafe displays Strict FIPS 140-2 Mode" = No in the information panel for that module.

The command line tool Enquiry does NOT include **StrictFIPS** in the list of flags for the module

## 6.3 Operating a level 2 module in FIPS mode

To be operating in Level 2 FIPS Mode, only FIPS Approved cryptography can be used in FIPS Mode. Use of any Non-FIPS Approved algorithms, except for those for which the CMVP allowed in FIPS Mode (See Supported Algorithms Section), means that the module would not be operating in FIPS Mode.

In order to comply with FIPS mode the operator must not generate private or secret keys with the **ExportAsPlain** ACL entry; nor should they use the **Import** service to import such keys in plain text.

An operator can verify that a key was generated correctly using the **nfmverify** utility supplied by Thales. This utility checks the ACL stored in the key-generation certificate.

## 6.4 To return a module to factory state

1. Put the mode switch into the initialisation position Pull the Initialisation pin high and restart the module.
2. Use the **Initialise** command to enter the Initialisation state.
3. Load a random value to use as the hash of the nShield Security Officer's key.
4. Set nShield Security Officer service to set the nShield Security Officer's key and the operational policy of the module.
5. Put the mode switch into the operational position Pull the Initialisation pin low and restart the module.
6. After this operation the module must be initialized correctly before it can be used in a FIPS approved mode.

Placing the module in factory state:

- destroys any loaded Logical tokens, Share Keys, Impath keys, Key objects, Session keys
- erases the current Module Signing Key and generates a fresh one.
- erases all current Module Keys, except the Well Known Module Key
- Generates a new Module Key Zero
- sets nShield Security Officer's key to a known value
- this prevents the module from loading any keys stored a key blobs as it no longer possesses the decryption key.

Returning the module to factory state does not erase the Firmware Confidentiality Key, the Long Term Signing Key or the public halves of the Firmware Integrity Key, of the Master Feature Enable Key: as these provide the cryptographic identity of the module and control loading firmware.

Thales supply a graphical user interface KeySafe and a command line tool **new-world** that automate these steps.

## 6.5 To create a new operator

1. Create a logical token.
2. Write one or more shares of this token onto software tokens.
3. For each key the operator will require, export the key as a key blob under this token.
4. Give the operator any pass phrases used and the key blob.

Thales supply a graphical user interface KeySafe and a command line tool **new-world** that automate these steps.

## 6.6 To authorize the operator to create keys

1. Create a new key, with an ACL that only permits **UseAsSigningKey**.
2. Export this key as a key blob under the operator's token.
3. Create a certificate signed by the nShield Security Officer's key that:
  - includes the hash of this key as the certifier.
  - authorizes the action **GenerateKey** or **GenerateKeyPair** depending on the type of key required.
4. If the operator needs to create permanent - as opposed to session - keys, the certificate must also include an entry that enables the action **MakeBlob**.  
The certificate can restrict the operator to only making blobs protected by their Operator Card Set by including the hash of its logical token.
5. Give the operator the key blob and certificate.

Thales supply a graphical user interface KeySafe and a command line tool **new-world** that automate these steps.

## 6.7 To authorize an operator to act as a Junior Security Officer

1. Generate a logical token to use to protect the Junior Security Officer's key.
2. Write one or more shares of this token onto software tokens
3. Create a new key pair,
4. Give the private half an ACL that permits **Sign** and **UseAsSigningKey**.
5. Give the public half an ACL that permits **ExportAsPlainText**
6. Export the private half of the Junior Security Officer's key as a key blob under this token.
7. Export the public half of the Junior Security Officer's key as plain text.
8. Create a certificate signed by the nShield Security Officer's key includes the hash of this key as the certifier

- authorizes the actions **GenerateKey**, **GenerateKeyPair**
  - authorizes the actions **GenerateLogicalToken**, **WriteShare** and **MakeBlob**, these may be limited to a particular module key.
9. Give the Junior Security Officer the software token, any pass phrases used, the key blob and certificate.

Thales supply a graphical user interface KeySafe and a command line tool **new-world** that automate these steps.

## 6.8 To authenticate an operator to use a stored key

1. Use the **LoadLogicalToken** service to create the space for a logical token.
2. Use the **ReadShare** service to read each share from the software token.
3. Use the **LoadBlob** service to load the key from the key blob.

The operator can now perform the services specified in the ACL for this key.

To assume nShield Security Officer role load the nShield Security Officer's key using this procedure. The nShield Security Officer's key can then be used in certificates authorising further operations.

Thales supply a graphical user interface KeySafe and a command line tool **new-world** that automate these steps.

## 6.9 To authenticate an operator to create a new key

1. If you have not already loaded your operator token, load it as above.
2. Use the **LoadBlob** service to load the authorization key from the key blob.
3. Use the **KeyId** returned to build a signing key certificate.
4. Present this certificate with the certificate supplied by the nShield Security Officer with the **GenerateKey**, **GenerateKeyPair** or **MakeBlob** command.

Thales supply a graphical user interface KeySafe and a command line tool **new-world** that automate these steps.

# Chapter 7: Physical security

All security critical components of the module are covered by epoxy resin.

The module has a clear button. Pressing this button put the module into the self-test state, clearing all stored key objects, logical tokens and impath keys and running all self-tests. The long term security critical parameters, nShield Security Officer's key, module keys and module signing key can be cleared by returning the module to the factory state, as described above.

## Checking the module

To ensure physical security, make the following checks regularly:

- Examine the epoxy resin security coating for obvious signs of damage.
- The smart card reader is directly plugged into the module or into a port provided by any appliance in which the module is integrated and the cable has not been tampered with. Where the module is in an appliance the security of this connection may be protected by the seals or other tamper evidence provided by the appliance.



# Chapter 8: Strength of functions

## Object IDs

Connections are identified by a **ClientID**, a random 32 bit number.

Objects are identified by an **ObjectID** again this is a random 32 bit number.

In order to randomly gain access to a key loaded by another operator you would need to guess two random 32 bit numbers. There are  $2^{64}$  possibilities therefore meets the 1 in a  $10^6$  requirement.

The module can process about  $2^{16}$  commands per minute - therefore the chance of succeeding within a minute is  $2^{16} / 2^{64} = 2^{-48}$  which is significantly less than the required chance of 1 in  $10^5$  ( $\sim 2^{-17}$ )

## Tokens

If an operator chooses to use a logical token with only one share, no pass phrase and leaves the smart card containing the share in the slot then another operator could load the logical token. The module does not have any idea as to which operator inserted the smart card. This can be prevented by:

- not leaving the smart card in the reader
- if the smart card is not in the reader, they can only access the logical token by correctly guessing the **ClientID** and **ObjectID** for the token.
- requiring a pass phrase
  - When loading a share requiring a pass phrase the operator must supply the SHA-1 hash of the pass phrase. The hash is combined with a module key, share number and smart card id to recreate the key used to encrypt the share. If the attacker has no knowledge of the pass phrase they would need to make  $2^{80}$  attempts to load the share. The module enforces a five seconds delay between failed attempts to load a share.
- requiring more than one share
  - If a logical token requires shares from more than one smart card the attacker would have to repeat the attack for each share required.

Logical tokens are either 168-bit Triple DES keys or 256-bit AES keys. Shares are encrypted under a combination of a module key, share number and card ID. If you could construct a logical token share of the correct form without knowledge of the module key and the exact mechanism used to derive the share key the chance that it would randomly decrypt into a valid token are  $2^{-168}$  or  $2^{-256}$ .

## Key Blobs

Key blobs are used to protect keys outside the module. There are two formats of blob - indirect and direct.

If the module is configured with AES module key, the blobs used for token and module key protected keys take a 256 bit AES key and a nonce and uses SHA-1 to derive a AES encryption key, used for encryption and a HMAC SHA-1 key, used for integrity.

If the module is configured with Triple DES module key, the blobs used for token and module key protected keys use Triple DES and SHA-1 for encryption and integrity.

If the module is initialised in a fresh security world, the blobs used for key-recovery and for pass-phrase recovery take the public half of a 3072-bit RSA key and a nonce as the input, and uses SHA-256 to derive a 256-bit AES encryption key, used for encryption and a HMAC SHA-256 key, used for integrity.

If the module is enrolled into an old security world created before the release of 2.50.16 firmware, the blobs used for key-recovery and for pass-phrase recovery take the public half of a 1024-bit RSA key and a nonce as the input, and uses SHA-1 to derive a 168-bit triple-DES or 256-bit AES encryption key - depending on the option selected for the module key - and a HMAC SHA-1 key, used for integrity.

The firmware also supports key blobs based on an integrated encryption scheme using Diffie Hellman to establish a master secret and HMAC SHA-256 for integrity and AES in CBC mode for encryption, or HMAC SHA-1 for integrity and Triple DES in CBC mode for encryption. However, this option is currently not used by any Thales tools.

All schemes used in SP800-131 compliant security worlds offer at least 128-bits of security. Those used in legacy security worlds offer at least 80-bits of security.

## Impaths

Impaths protect the transfer of encrypted shares between modules.

When negotiating an Impath, provided both modules use 2.50.16 or later firmware, the module verifies a 3072-bit DSA signatures with SHA-256 hashes to verify the identity of the other module. It then uses 3072-bit Diffie-Hellman key exchange to negotiate a 256-bit AES encryption and MAC keys used to protect the channel. This provides a minimum of 128-bits of security for the encrypted channel.

Otherwise, both modules use 1024-bit DSA signatures to verify the identity of the other module. Then they perform a 1024-bit Diffie-Hellman key exchange to negotiate a 168-bit triple-DES encryption keys used to protect the channel. This provides a minimum of 80-bits of security for the encrypted channel.

**Note:** The shares sent over the channel are still encrypted using their share key, decryption only takes place on the receiving module.

## KDP key provisioning

The KDP protocol used to transfer keys from a module to a micro HSM uses 1024-bit DSA signatures to identify the end point and a 2048-bit Diffie-Hellman key exchange to negotiate the

Triple-DES or AES keys used to encrypt the keys in transit providing a minimum of 100-bits of security for the encrypted channel.

## Derived Keys

The nCore API provides a range of key derivation and wrapping options that an operator can choose to make use of in their protocols.

For any key, these mechanisms are only available if the operator explicitly enabled them in the key's ACL when they generated or imported the key.

The ACL can specify not only the mechanism to use but also the specific keys that may be used if these are known.

| Mechanism                  | Use   | Notes  |
|----------------------------|---|--|
| Key Splitting              | Splits a symmetric key into separate components for export                    | Components are raw byte blocks.  |
| PKCS8 wrapping             | Encrypts a key using a pass phrase.   | Only available in FIPS 140-2 level 2 mode  |
| PKCS8 unwrapping           | Decrypts a wrapped key using a pass phrase.                                   | Only available in FIPS 140-2 level 2 mode  |
| SSL3 master key derivation | Setting up a SSL session  | Only available in FIPS 140-2 level 2 mode  |
| TLS master key derivation  | Setting up a TLS session  |  |
| Key Wrapping               | Encrypts one key object with another to allow the wrapped key to be exported. | May use any supported encryption mechanism that accepts a byteblock. The operator must ensure that they chose a wrapping key that has an equivalent strength to the key being transported. |

In level 2 mode you can use key wrapping and key establishment mechanisms with all supported algorithms.

# Chapter 9: Self Tests

When power is applied to the module it enters the self test state. The module also enters the self test state whenever the unit is reset, by pressing the clear button.

In the self test state the module clears the main RAM, thus ensuring any loaded keys or authorization information is removed and then performs its self test sequence, which includes:

- An operational test on hardware components
- An integrity check on the firmware, verification of a SHA-1 hash.
- A statistical check on the random number generator
- Known answer checks as required by FIPS 140-2.
- Verification of a MAC on EEPROM contents to ensure it is correctly initialized.

This sequence takes a few seconds after which the module enters the Pre-Maintenance, Pre-Initialisation, Uninitialised or Operational state; depending on the position of the mode switch and the validity of the EEPROM contents.

The module also continuously monitors the hardware entropy source and the approved AES-256 based DRBG. If either fail it enters the error state.

When firmware is updated, the module verifies a DSA signature on the new firmware image before it is written to flash.

The module also performs pairwise-consistency checks when generating asymmetric key-pairs.

In the error state, the module's LED repeatedly flashes the Morse pattern SOS, followed by a status code indicating the error. All other inputs and outputs are disabled.

## 9.1 Firmware Load Test

When new firmware is loaded, the module reads the candidate image into working memory. It then performs the following tests on the image before it replaces the current application:

- The image contains a valid signature which the module can verify using the Firmware Integrity Key
- The image is encrypted with the Firmware Confidentiality Key stored in the module.
- The Version Security Number for the image is at least as high as the stored value.

Only if all three tests pass is the new firmware written to permanent storage.

Updating the firmware clears the nShield Security Officer's key and all stored module keys. The module will not re-enter operational mode until the administrator has correctly re-initialized it.

# Chapter 10: Supported Algorithms

## 10.1 FIPS approved and allowed algorithms:

### Symmetric Encryption

#### AES

Certificate #2122

ECB, CBC, GCM and CMAC modes

#### Triple-DES

Certificate #1349

ECB and CBC modes

### Hashing and Message Authentication

#### AES CMAC

AES Certificate #2122

#### AES GMAC

AES Certificate #2122

#### HMAC SHA-1, HMAC SHA-224, HMAC SHA-256, HMAC SHA-384 and HMAC SHA-512

Certificate #1292

#### SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512

Certificate #1844

#### Triple-DES MAC

Triple-DES Certificate #1349 vendor affirmed

### Signature

#### DSA

Certificate #664 (all models) and Certificate #777 (500+, 1500+, 6000+ models)

FIPS 186-2 and FIPS 186-4: Signature generation and verification (The module supports the use of keys between 1,024 bits and 3,072 bits. Signature generation is non-compliant for SHA-1, and for keys less than 2,048 bits.)

**ECDSA**

Certificate #318 (10+ model), Certificate #181 (500+, 1500+, 6000+ models)

FIPS186-2: Signature generation and verification (Signature generation is non-compliant for SHA-1, and for values of n less than 224 bits, and for the curves P-192 K-163 and B-163.)

P-192 P-224 P-256 P-384 P-521 K-163 K-233 K-283 K-409 K-571 B-163 B-233 B-283 B-409 and B-571 Curves

**RSA**

Certificate #1092 (all models) and Certificate #1299 (500+, 1500+, 6000+ models)

FIPS 186-4: Key generation; RSASSA PKCS1\_V1\_5 and RSASSA-PSS signature generation and verification (Signature generation is non-compliant for SHA-1, and for keys less than 2,048 bits.)

Modulus 1024 - 4096 bits with SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512

**Key Establishment****Diffie-Hellman (for 500+, 1500+ and 6000+ models only)**

Diffie-Hellman (CVL Certs. #27 and #90, key agreement; key establishment methodology provides between 112 and 256 bits of encryption strength; non-compliant less than 112 bits of encryption strength)

**Elliptic Curve Diffie-Hellman (for 500+, 1500+ and 6000+ models only)**

EC Diffie-Hellman (CVL Certs. #27 and #90, key agreement; key establishment methodology provides between 112 and 256 bits of encryption strength; non-compliant less than 112 bits of encryption strength)

**Diffie-Hellman (for 10+ only)**

Diffie-Hellman (key agreement; key establishment methodology provides between 112 and 256 bits of encryption strength; non-compliant less than 112 bits of encryption strength)

**Elliptic Curve Diffie-Hellman (for 10+ only)**

EC Diffie-Hellman (key agreement; key establishment methodology provides between 112 and 256 bits of encryption strength; non-compliant less than 112 bits of encryption strength)

**RSA**

RSA (key wrapping, key establishment methodology provides between 112 and 256 bits of encryption strength; non-compliant less than 112 bits of encryption strength)

**AES**

AES (Cert. #2122, key wrapping; key establishment methodology provides between 128 and 256 bits of encryption strength)

AES Key Wrap, AES CMAC Counter mode according to SP800-108, AES CBC mode

### **Triple-DES**

Triple-DES (Cert. #1349, key wrapping; key establishment methodology provides 112 bits of encryption strength; non-compliant less than 112 bits of encryption strength)

CBC mode

### **Other**

#### **Deterministic Random Bit Generator**

Certificate #232

SP 800-90 using Counter mode of AES-256

#### **EC-MQV**

ECMQV (key agreement; key establishment methodology provides between 112 and 256 bits of encryption strength; non-compliant less than 112 bits of encryption strength)

## 10.2 Non-FIPS approved algorithms

**Note:** Algorithms marked with an asterisk are not approved by NIST. In level 2 mode, the algorithms are available. However, if you choose to use them, the module is not operating in FIPS approved mode.

### Symmetric

- Aria<sup>\*</sup>
- Arc Four (compatible with RC4)<sup>\*</sup>
- Camellia<sup>\*</sup>
- CAST 6 (RFC2612)<sup>\*</sup>
- DES<sup>\*</sup>
- SEED (Korean Data Encryption Standard) - requires Feature Enable activation<sup>\*</sup>

### Asymmetric

- El Gamal<sup>\*</sup> (encryption using Diffie-Hellman keys)
- KCDSA (Korean Certificate-based Digital Signature Algorithm) - requires Feature Enable activation<sup>\*</sup>
- RSA encryption and decryption<sup>\*</sup> (Same RSA implementation as used for key wrapping)

### Hashing and Message Authentication

- HAS-160 - requires Feature Enable activation<sup>\*</sup>
- MD5 - requires Feature Enable activation<sup>\*</sup>
- RIPEMD 160<sup>\*</sup>
- Tiger<sup>\*</sup>
- HMAC (MD5, RIPEMD160, Tiger)<sup>\*</sup>

### Non-deterministic entropy source

Non-deterministic entropy source, used to seed approved random bit generator.

### Other

SSL<sup>\*</sup>/TLS master key derivation

PKCS #8 padding<sup>\*</sup>.

**Note:** TLS key derivation is approved for use by FIPS 140-2 validated modules - though there is as yet no validation test. MD5 may be used within TLS key derivation.



# Addresses

## Americas

900 South Pine Island Road, Suite 710, Plantation, Florida 33324, USA

Tel: +1 888 744 4976 or + 1 954 888 6200

[sales@thalessec.com](mailto:sales@thalessec.com)

## Europe, Middle East, Africa

Meadow View House, Long Crendon, Aylesbury, Buckinghamshire HP18 9EQ, UK

Tel: + 44 (0)1844 201800

[emea.sales@thales-esecurity.com](mailto:emea.sales@thales-esecurity.com)

## Asia Pacific

Units 4101, 41/F. 248 Queen's Road East, Wanchai, Hong Kong, PRC

Tel: + 852 2815 8633

[asia.sales@thales-esecurity.com](mailto:asia.sales@thales-esecurity.com)

## Internet addresses

Web site: <http://www.thales-esecurity.com/>  
Support: <http://www.thales-esecurity.com/support-landing-page>  
Online documentation: <http://www.thales-esecurity.com/knowledge-base>  
International sales offices: <http://www.thales-esecurity.com/contact>