# SecureAgent® Software Cryptographic Module

Version 2.2.006
Security Policy

**FIPS 140-2 Level 1 Validation**

**September 5, 2013**
**Version 1.24**

© 2013, SecureAgent® Software Inc. All rights reserved. www.SecureAgent.com
This document may be freely reproduced and distributed whole and intact including this copyright notice.

1 Cryptographic Module Specification ..... 3

2 Cryptographic Module Ports and Interfaces ..... 7

3 Roles, Services, and Authentication ..... 8

4 Finite State Model ..... 28

5 Physical Security (N/A) ..... 28

6 Operational Environment ..... 28

7 Cryptographic Key Management ..... 29

8 Electromagnetic Interference/Electromagnetic Compatibility (EMI/EMC) ..... 30

9 Self-Tests ..... 30

10 Design Assurance ..... 30

11 Mitigation of Attacks (N/A) ..... 31

References ..... 31

# 1 Cryptographic Module Specification

This document is the Security Policy for the SecureAgent® Software Cryptographic Module. This security policy specifies the security rules under which the module shall operate to meet the requirements of FIPS 140-2 Level 1. It describes how the module functions to meet the FIPS requirements, and the actions that operators must take to maintain the security of the module.

This security policy describes the features and design of the SecureAgent® Software Cryptographic Module (a shared library providing cryptographic services and security functions) using the terminology contained in the FIPS 140-2 specification. *FIPS 140-2, Security Requirements for Cryptographic Modules* specifies the security requirements that must be satisfied by a cryptographic module utilized within a security system protecting sensitive but unclassified information. The NIST Cryptographic Module Validation Program (CMVP) validates cryptographic modules to FIPS 140-2 standards. Validated products are accepted by the Federal agencies of both the USA and Canada for the protection of sensitive or designated information.

For information on the FIPS 140-2 standard and the CMVP program, visit http://csrc.nist.gov/groups/STM/cmvp. More information describing the SecureAgent® Software Cryptographic Module can be found at http://www.SecureAgent.com.

This security policy contains only non-proprietary information. All other documentation submitted for FIPS 140-2 conformance testing and validation is “SecureAgent® Software - Proprietary” and is releasable only under appropriate non-disclosure agreements.

The SecureAgent® Software Cryptographic Module meets the overall requirements applicable to Level 1 security for FIPS140-2.

***Table 1. Cryptographic Module Security Requirements***

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

### 1.1 Document History

**Table 2. Document Version**

| Version | Date | Comments | Name |
|---|---|---|---|
| 1.00 | 9/13/2007 | Initial Draft | Ward Rosenberry |
| 1.06 | 02/11/2011 | Update | Edwin Woehrstein |
| 1.07 | 03/17/2011 | Update | Edwin Woehrstein |
| 1.08 | 04/07/2011 | Updated with TOR2/3 information | Edwin Woehrstein |
| 1.09 | 04/07/2011 | Update | Steve Soodsma |
| 1.10 | 07/25/2011 | Updated with TOR4/5 information | Steve Soodsma |
| 1.11 | 08/31/2011 | Updated with TOR6 information | Steve Soodsma |
| 1.12 | 09/01/2011 | Added Zeroization service to Table 5 | Steve Soodsma |
| 1.13 | 06/06/2012 | Updated module name to libsai2s_x86.so<br>Updated api function names<br>Added HMAC, SHS, and RNG to table 5 | Steve Soodsma |
| 1.14 | 06/25/2012 | Updated with TOR7/8 information | Steve Soodsma |
| 1.15 | 07/18/2012 | Updated with TOR9 information | Steve Soodsma |
| 1.16 | 07/19/2012 | Updated table 7 | Steve Soodsma |
| 1.17 | 07/25/2012 | Updated with TOR10 information | Steve Soodsma |
| 1.18 | 02/19/2013 | Addressed received comments | Steve Soodsma |
| 1.19 | 04/03/2013 | Added tables 5.3 and 5.4 | Steve Soodsma |
| 1.20 | 04/12/2013 | Updated sections 1.4, 3.2, and Table 5.3<br>Removed Table 5.4 | Steve Soodsma |
| 1.21 | 04/19/2013 | Added N/A to section 1.2<br>Added module version to section 1.4 | Steve Soodsma |
| 1.22 | 05/15/2013 | Updated sections 1.4 and 9.1 | Steve Soodsma |
| 1.23 | 09/04/2013 | Updated sections 1.5 and 3.2 | Steve Soodsma |
| 1.24 | 09/05/2013 | Updated sections 1.5 and 3.2. Corrected typos. | Steve Soodsma |

### 1.2 Acronyms and Abbreviations

| | |
|---|---|
| AES | Advanced Encryption Standard |
| CBC | Cipher Block Chaining |
| CFB | Cipher Feedback |
| CMVP | Cryptographic Module Validation Program |
| CSE | Communications Security Establishment |
| CSP | Critical Security Parameter |
| DLL | Dynamic Linked Library |
| EK | Encryption Key |
| EMC | Electromagnetic Compatibility |
| EMI | Electromagnetic Interference |
| FCC | Federal Communication Commission |
| FIPS | Federal Information Processing Standard |

| | |
|---|---|
| HEX | Hexadecimal representation of a binary value |
| HMAC | Keyed-Hash Message Authentication Code |
| KAT | Known Answer Test |
| KEK | Key Encryption Key |
| LAN | Local Area Network |
| NIST | National Institute of Standards and Technology |
| PRNG | Pseudo Random Number Generator |
| PUB | Publication |
| SHA-1 | Secure Hash Algorithm |
| N/A | Not Applicable |

## *1.3 Functional Overview*

The SecureAgent® Software Cryptographic Module provides the core cryptographic services for several secure communications and controller systems designed and manufactured by SecureAgent® Software.

The SecureAgent® Software Cryptographic Module consists of functions that AES-encrypt data for the Application that is being written to an unprotected network or storage medium. Conversely the SecureAgent® Software Cryptographic Module decrypts AES-encrypted data that the Application reads from an unprotected network or storage medium and outputs the plaintext data for use within a protected environment. The SecureAgent® Software Cryptographic Module forms a cohesive subsystem within the SecureAgent Application that is contained in a larger controlling system (such as complete solution devices) that is outside the evaluation scope of this module.

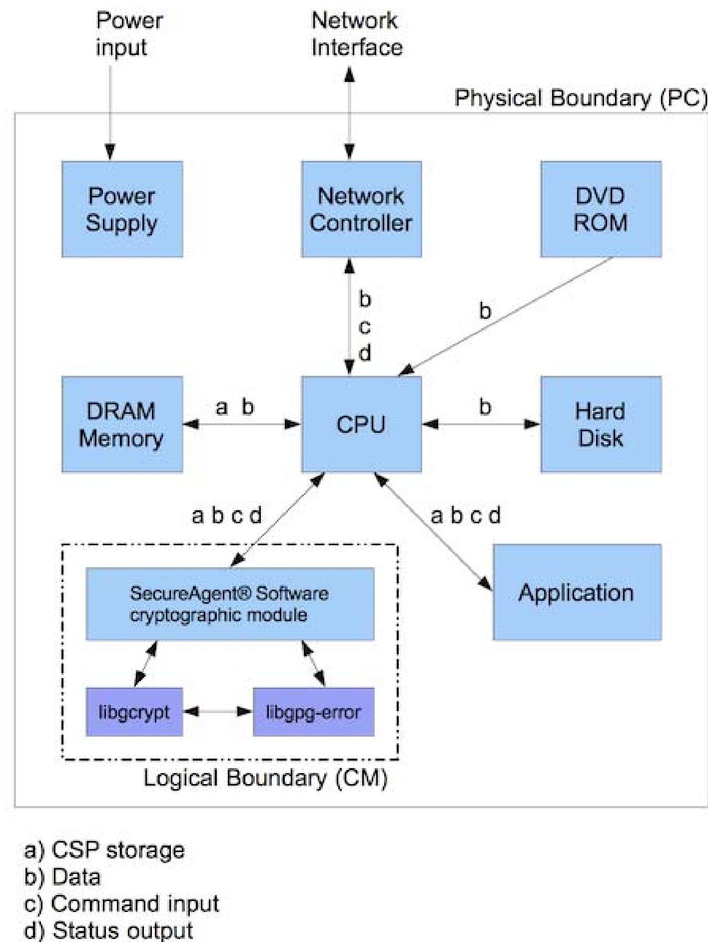
Features of the software include:

- AES (128, 192, or 256-bit) data encryption
- FIPS 140-2 power-on self-test and conditional test.

Figure 1 illustrates typical module implementation. The SecureAgent® Software Cryptographic Module provides API functions to

- Encrypt a data block
- Decrypt a data block
- Compress / Decompress a data block
- Encrypt and convert to HEX representation
- Decrypt a HEX representation of a data block
- Test the module

***Figure 1. High Level Functional View of the SecureAgent® Software Cryptographic Module Function***




## 1.4 Module Description

The SecureAgent® Software Cryptographic Module is a software cryptographic module that executes within the SecureAgent Application. The SecureAgent® Software Cryptographic Module is tested on Sun Solaris 10.

The SecureAgent® Software Cryptographic Module provides data encryption and decryption services, compression/decompression services and a service to convert the data after encryption into a human readable HEX representation and also to decrypt a human readable HEX representation of encrypted data. Software integrity services assure operators of a valid state within the module. The SecureAgent® Software Cryptographic Module does not have a bypass or maintenance mode. The module is in FIPS mode after calling sai_init to initialize the module and while using services listed in table 5.1.

- A non-Approved mode exists.
- The services available for the non-Approved mode are listed in Table 5.3.
- The module is configured during initialization to operate only in FIPS mode by using the services listed in Table 5.1 when in the operational state.

The files that comprise the SecureAgent® Software Cryptographic Module version 2.2.006 are:

libsai2s_x86.so
libgcrypt.so.11.7.0
libgpg-error.so.0.8.0

### 1.5 Module Security Functions

The SecureAgent® Software Cryptographic Module implements the security functions described in Table 3. The Algorithms listed in Table3 are only available in the Approved mode. The Algorithms listed in Table 3a are only available in the non-Approved mode.

***Table 3. Module Security Functions.***

| *Approved Security Function* | *Certificate* |
|---|---|
| **AES** (FIPS PUB 197) | #2044 |
| **SHA-256** (FIPS PUB 180-2) | #1790 |
| **HMAC-SHA256** (FIPS PUB 198a) | #1243 |
| **RNG** (ANSI X9.31 Appendix A.2.4 using AES) | #1067 |
| **NDRNG** (Sun Solaris 10 /dev/random used for seeding) | |

***Table 3a. Module Non-Approved Mode Algorithms.***

| | |
|---|---|
| Symmetric Key | |
| AES128, non-compliant | Message Digests and HMAC |
| AES192, non-compliant | HAVAL |
| AES256, non-compliant | MD2 |
| ARCFOUR | MD4 |
| BLOWFISH | MD5 |
| CAMELLIA128 | RMD160 |
| CAMELLIA192 | SHA1, non-compliant |
| CAMELLIA256 | SHA224, non-compliant |
| CAST5 | SHA256, non-compliant |
| DES | SHA384, non-compliant |
| RC2 | SHA512, non-compliant |
| SEED | TIGER |
| SERPENT128 | TIGER1 |
| SERPENT192 | TIGER2 |
| SERPENT256 | WHIRLPOOL |
| TDES | Random Number Generator |
| TWOFISH | X9.31 RNG, non-compliant |
| TWOFISH128 | Key Derivation |
| Asymmetric Key | SIMPLE_S2K |
| RSA, non-compliant | SALTED_S2K |
| DSA, non-compliant | ITERSALTED_S2K |
| Elgamal | PBKDF2 |

## 2 Cryptographic Module Ports and Interfaces

The SecureAgent® Software Cryptographic Module meets the requirements of a multi-chip standalone module. Since the SecureAgent® Software Cryptographic Module is a software module, its interfaces are defined in terms of the API that it provides. Data Input Interface is defined as the input data parameters of those API functions that accept, as their arguments, data to be used or processed by the module. The return value or arguments of appropriate types, data generated or otherwise processed by the API functions to the caller constitute Data Output Interface. Control Input Interface is comprised of the call used to initiate the module and the API functions used to control the operation of the module. Status Output Interface is defined as the API function sai_get_status that provides information about the status of the module. The function sai_get_status may be called anytime to indicate the status of the SecureAgent® Software Cryptographic Module.

***Table 4. Module Ports and Interfaces.***

| **FIPS 140-2 Interface** | **Logical Interface** | **Module Physical Interface** |
|---|---|---|
| **Data Input** | API input parameters | N/A |

| FIPS 140-2 Interface | Logical Interface | Module Physical Interface |
|---|---|---|
| **Data Output** | API output parameters | N/A |
| **Control Input** | API function calls, or configuration files on filesystem | N/A |
| **Status Output** | sai_get_status | N/A |
| **Power Input** | N/A | N/A |

# 3 Roles, Services, and Authentication

## 3.1 Roles

The two roles are defined per the FIPS140-2 standard as follows:

1. **Crypto Officer** - can access all services implemented in the Module. The Crypto Officer can also install the Module on the target operating system (Sun Solaris 10) and configure the operating system for Module use. No special access to keys or data is provided to this role. The Crypto Officer role is implicitly selected when installing the module or configuring the operating system for the Module.
2. **User** - can access all services implemented in the Module. This role is implicitly selected when an application calls any of the API functions in the Module.

In the SecureAgent® Software Cryptographic Module, an operator is implicitly assumed in the User or Cryptographic Officer role based upon the operations chosen. Both User and Cryptographic Officer can call all services implemented in the Module.

The user role is assumed by the subsystem writing plaintext data to or reading plaintext data from the module after it has passed the FIPS power-on self-test. These actions comprise the module encryption and decryption services.

Multiple concurrent operators are not allowed. Only a single user may access the module at any given point in time. Operators cannot change roles while using the module. The strength of the operator authentication, per the above roles, does not apply to this cryptographic module as it occurs outside of the module.

## 3.2 Services

Tables 5.1 and 5.2 show the services available to the various roles. Encrypt and decrypt services delete the key from memory when the operation completes without modifying, disclosing, or substituting the key in any manner. Table 5.3 lists the libgcrypt services that are available in the non-Approved mode of operation.

- CSPs defined in the FIPS mode **shall** not be accessed or shared while in the non-FIPS mode.
- CSPs **shall** not be generated while in the non-FIPS mode and then accessed or shared while in the FIPS mode.
- The Approved RNG may be used in the non-FIPS mode but the Approved RNG's seed or seed key **shall** not be accessed or shared in the non-FIPS mode.

***Table 5.1 Roles and Services***

| *Service* | *Description* | *Crypto Officer* | *User* | *Inputs* | *Outputs* |
|---|---|---|---|---|---|
| sai_encrypt | Encrypts plain text input | ● | ● | Plain text, key | Cipher text |
| sai_decrypt | Decrypts cipher text input | ● | ● | Cipher text, key | Plain text |
| sai_cipher_open<br>sai_cipher_setiv<br>sai_cipher_setkey<br>sai_cipher_decrypt | Used for AES algorithm validation. The MCT test performs CBC chaining in the test program | ● | ● | Plain text, Cipher text, key | Cipher-hd, Cipher text, Plain text |

| Service | Description | Crypto Officer | User | Inputs | Outputs |
|---|---|---|---|---|---|
| sai_cipher_encrypt<br>sai_cipher_close | dictating the need for these lower-level functions. | | | | |
| sai_hmac256_new<br>sai_hmac256_update<br>sai_hmac256_finalize<br>sai_hmac256_release | Used for integrity check | ● | ● | Data, key | HMAC-SHA-256 |
| sai_hmac256_new(NULL Key)<br>sai_hmac256_update<br>sai_hmac256_finalize<br>sai_hmac256_release | Used in generating SHA-256 | ● | ● | Data | SHA-256 |
| sai_create_nonce | Generates random numbers for use in AES-CBC initialization vectors | ● | ● | No input | Nonce |
| sai_version | Returns module version number | ● | ● | No input | Module version, FIPS Mode |
| sai_selftest | Performs module self-test | ● | ● | No input | Return code Pass/Fail |
| sai_get_status | Returns module's current status | ● | ● | Message buffer | FSM State, Self Test status |
| sai_init | Initializes FSM, performs self-tests, and module integrity check, FIPS Mode | ● | ● | No input | Self Test status FIPS Mode |
| sai_zeroize | Safely zeroizes memory | ● | ● | Key or CSP to zeroize | No output |

**Table 5.2 Auxillary Functions**

| Service | Description | Crypto Officer | User | Inputs | Outputs |
|---|---|---|---|---|---|
| sai_encrypt_hex | Utility function for debug. Encrypts a data block to a hex representation. | ● | ● | Data, key, IV | Hex representation of encrypted data |
| sai_decrypt_hex | Utility function for debug. Decrypts a hex encrypted data block to binary. | ● | ● | Hex data, key, IV | Binary decrypted data |
| sai_rngfips_run_external_test<br>sai_rngfips_deinit_external_test<br>sai_rngfips_init_external_test | Used for RNG algorithm validation. Allows to test RNG with a Known Answer Test. | ● | ● | RNG KAT values | Status code |
| sai_convert_bin_to_hex | Utility function to convert binary data to hex representation | ● | ● | Binary data | Hex data |
| sai_convert_hex_to_bin | Utility function to convert hex data to binary representation. | ● | ● | Hex data | Binary data |
| sai_strerror<br>sai_strsource | Utility functions to convert error codes to human-readable string | ● | ● | Error code | String Error message |

| ***Service*** | ***Description*** | ***Crypto Officer*** | ***User*** | ***Inputs*** | ***Outputs*** |
|---|---|---|---|---|---|
| | messages. | | | | |
| sai_malloc<br>sai_free | General purpose memory allocation. | ● | ● | Size | Memory block |
| sai_compress<br>sai_decompress | Utility functions to provide data compression and decompression services | ● | ● | Source data, output buffer, work buffer | Output data |

**Table 5.3 Libgcrypt Services available in non-Approved mode**

| ***Service*** | ***Description*** | ***Crypto Officer*** | ***User*** | ***Inputs*** | ***Outputs*** |
|---|---|---|---|---|---|
| gcry_err_make<br>gcry_error<br>gcry_err_code<br>gcry_err_source | Wrappers for the libgpg-error library. | ● | ● | (source, code)<br>(code)<br>(err)<br>(err) | gcry_error_t<br>gcry_error_t<br>gcry_err_code_t<br>gcry_err_source_t |
| gcry_strerror | Return a pointer to a string containing a description of the error code in the error value ERR. | ● | ● | (err) | const char * |
| gcry_strsource | Return a pointer to a string containing a description of the error source in the error value ERR. | ● | ● | (err) | const char * |
| gcry_err_code_from_errno | Retrieve the error code for the system error ERR. This returns GPG_ERR_UNKNOWN_ERRNO if the system error is not mapped (report this). | ● | ● | (err) | gcry_err_code_t |
| gcry_err_code_to_errno | Retrieve the system error for the error code CODE. This returns 0 if CODE is not a system error code. | ● | ● | (code) | int |
| gcry_err_make_from_errno | Return an error value with the error source SOURCE and the system error ERR. | ● | ● | (source, err) | gcry_error_t |
| gcry_error_from_errno | Return an error value with the system error ERR. | ● | ● | (err) | gcry_err_code_t |
| gcry_pth_select<br>gcry_pth_waitpid<br>gcry_pth_accept<br>gcry_pth_connect | GNU Pth network wrapper macro | ● | ● | (int nfd, fd_set *rset, fd_set *wset, set *eset, struct timeval *timeout)<br>(pid_t pid, int *status, int options)<br>(int s, struct sockaddr *addr, gcry_socklen_t *length_ptr)<br>(int s, struct sockaddr *addr, gcry_socklen_t length) | ssize_t<br>ssize_t<br>int<br>int |
| gcry_pthread_mutex_init<br>gcry_pthread_mut | pthread wrapper macros | ● | ● | (void **priv)<br>(void **lock)<br>(void **lock) | int<br>int<br>int |

| | | | | | |
|---|---|---|---|---|---|
| ex_destroy<br>gcry_pthread_mutex_lock<br>gcry_pthread_mutex_unlock | | | | (void **lock) | int |
| gcry_check_version | Check that the library fulfills the version requirement. | ● | ● | (const char *req_version) | const char * |
| gcry_control | Perform various operations defined by CMD. | ● | ● | (enum gcry_ctl_cmds CMD, ...) | gcry_error_t |
| gcry_sexp_new | Create an new S-expression object from BUFFER of size LENGTH and return it in RETSEXP. With AUTODETECT set to 0 the data in BUFFER is expected to be in canonized format. | ● | ● | (gcry_sexp_t *retsexp, const void *buffer, size_t length, int autodetect) | gcry_error_t |
| gcry_sexp_create | Same as gcry_sexp_new but allows to pass a FREEFNC which has the effect to transfer ownership of BUFFER to the created object. | ● | ● | (gcry_sexp_t *retsexp, void *buffer, size_t length, int autodetect, void (*freefnc) (void *)) | gcry_error_t |
| gcry_sexp_sscan | Scan BUFFER and return a new S-expression object in RETSEXP.This function expects a printf like string in BUFFER. | ● | ● | (gcry_sexp_t *retsexp, size_t *erroff, const char *buffer, size_t length) | gcry_error_t |
| gcry_sexp_build | Same as gcry_sexp_sscan but expects a string in FORMAT and can thus only be used for certain encodings. | ● | ● | (gcry_sexp_t *retsexp, size_t *erroff, const char *format, ...) | gcry_error_t |
| gcry_sexp_build_array | Like gcry_sexp_build, but uses an array instead of variable function arguments. | ● | ● | (gcry_sexp_t *retsexp, size_t *erroff, const char *format, void **arg_list) | gcry_error_t |
| gcry_sexp_release | Release the S-expression object SEXP | ● | ● | (gcry_sexp_t sexp) | void |
| gcry_sexp_canon_len | Calculate the length of an canonized S-expresion in BUFFER and check for a valid encoding. | ● | ● | (const unsigned char *buffer, size_t length, size_t *erroff, gcry_error_t *errcode) | size_t |
| gcry_sexp_sprint | Copies the S-expression object SEXP into BUFFER using the format specified in MODE. | ● | ● | (gcry_sexp_t sexp, int mode, void *buffer, size_t maxlength) | size_t |
| gcry_sexp_dump | Dumps the S-expression object A in a format suitable for debugging to Libgcrypt's logging stream. | ● | ● | (const gcry_sexp_t a) | void |
| gcry_sexp_cons<br>gcry_sexp_alist<br>gcry_sexp_vlist<br>gcry_sexp_append<br>gcry_sexp_prepend | General S-expression functions used with the public key functions. | ● | ● | (const gcry_sexp_t a, const gcry_sexp_t b)<br>(const gcry_sexp_t *array)<br>(const gcry_sexp_t a, …)<br>(const gcry_sexp_t a, const gcry_sexp_t n)<br>(const gcry_sexp_t a, | gcry_sexp_t<br>gcry_sexp_t<br>gcry_sexp_t<br>gcry_sexp_t<br>gcry_sexp_t |

| | | | | | |
|---|---|---|---|---|---|
| | | | | const gcry_sexp_t n); | |
| gcry_sexp_find_token | Scan the S-expression for a sublist with a type (the car of the list) matching the string TOKEN. If TOKLEN is not 0, the token is assumed to be raw memory of this length. The function returns a newly allocated S-expression consisting of the found sublist or `NULL' when not found. | ● | ● | (gcry_sexp_t list, const char *tok, size_t toklen) | gcry_sexp_t |
| gcry_sexp_length | Return the length of the LIST. For a valid S-expression this should be at least 1. | ● | ● | (const gcry_sexp_t list) | int |
| gcry_sexp_nth | Create and return a new S-expression from the element with index NUMBER in LIST. Note that the first element has the index 0. If there is no such element, `NULL' is returned. | ● | ● | (const gcry_sexp_t list, int number) | gcry_sexp_t |
| gcry_sexp_car | Create and return a new S-expression from the first element in LIST; this called the "type" and should always exist and be a string. `NULL' is returned in case of a problem. | ● | ● | (const gcry_sexp_t list) | gcry_sexp_t |
| gcry_sexp_cdr | Create and return a new list form all elements except for the first one. Note, that this function may return an invalid S-expression because it is not guaranteed, that the type exists and is a string. However, for parsing a complex S-expression it might be useful for intermediate lists. Returns `NULL' on error. | ● | ● | (const gcry_sexp_t list) | gcry_sexp_t |
| gcry_sexp_cadr | | ● | ● | (const gcry_sexp_t list) | gcry_sexp_t |
| gcry_sexp_nth_data | This function is used to get data from a LIST. A pointer to the actual data with index NUMBER is returned and the length of this data will be stored to DATALEN. If there is no data at the given index or the index represents another list, `NULL' is returned. *Note:* The returned pointer is valid as long as LIST is not modified or released. | ● | ● | (const gcry_sexp_t list, int number, size_t *datalen) | const char * |

| gcry_sexp_nth_string | This function is used to get and convert data from a LIST. The data is assumed to be a Null terminated string. The caller must release the returned value using `gcry_free'. If there is no data at the given index, the index represents a list or the value can't be converted to a string, `NULL' is returned. | ● | ● | (gcry_sexp_t list, int number) | char * |
|---|---|---|---|---|---|
| gcry_sexp_nth_mpi | This function is used to get and convert data from a LIST. This data is assumed to be an MPI stored in the format described by MPIFMT and returned as a standard Libgcrypt MPI. The caller must release this returned value using `gcry_mpi_release'. If there is no data at the given index, the index represents a list or the value can't be converted to an MPI, `NULL' is returned. | ● | ● | (gcry_sexp_t list, int number, int mpifmt); | gcry_mpi_t |
| gcry_mpi_new | Allocate a new big integer object, initialize it with 0 and initially allocate memory for a number of at least NBITS. | ● | ● | (unsigned int nbits) | gcry_mpi_t |
| gcry_mpi_snew | Same as gcry_mpi_new() but allocate in "secure" memory. | ● | ● | (unsigned int nbits) | gcry_mpi_t |
| gcry_mpi_release | Release the number A and free all associated resources. | ● | ● | (gcry_mpi_t a) | void |
| gcry_mpi_copy | Create a new number with the same value as A. | ● | ● | (const gcry_mpi_t a) | gcry_mpi_t |
| gcry_mpi_set | Store the big integer value U in W. | ● | ● | (gcry_mpi_t w, const gcry_mpi_t u) | gcry_mpi_t |
| gcry_mpi_set_ui | Store the unsigned integer value U in W. | ● | ● | (gcry_mpi_t w, unsigned long u) | gcry_mpi_t |
| gcry_mpi_swap | Swap the values of A and B. | ● | ● | (gcry_mpi_t a, gcry_mpi_t b) | void |
| gcry_mpi_cmp | Compare the big integer number U and V returning 0 for equality, a positive value for U > V and a negative for U < V. | ● | ● | (const gcry_mpi_t u, const gcry_mpi_t v) | int |
| gcry_mpi_cmp_ui | Compare the big integer number U with the unsigned integer V returning 0 for equality, a positive value for U > V and a negative for U < V. | ● | ● | (const gcry_mpi_t u, unsigned long v) | int |

| | | | | | |
|---|---|---|---|---|---|
| gcry_mpi_scan | Convert the external representation of an integer stored in BUFFER with a length of BUFLEN into a newly create MPI returned in RET_MPI. If NSCANNED is not NULL, it will receive the number of bytes actually scanned after a successful operation. | ● | ● | (gcry_mpi_t *ret_mpi, enum gcry_mpi_format format, const void *buffer, size_t buflen, size_t *nscanned); | gcry_error_t |
| gcry_mpi_print | Convert the big integer A into the external representation described by FORMAT and store it in the provided BUFFER which has been allocated by the user with a size of BUFLEN bytes. NWRITTEN receives the actual length of the external representation unless it has been passed as NULL. | ● | ● | (enum gcry_mpi_format format, unsigned char *buffer, size_t buflen, size_t *nwritten, const gcry_mpi_t a) | gcry_error_t |
| gcry_mpi_aprint | Convert the big integer A int the external representation described by FORMAT and store it in a newly allocated buffer which address will be put into BUFFER. NWRITTEN receives the actual lengths of the external representation. | ● | ● | (enum gcry_mpi_format format, unsigned char **buffer, size_t *nwritten, const gcry_mpi_t a) | gcry_error_t |
| gcry_mpi_dump | Dump the value of A in a format suitable for debugging to Libgcrypt's logging stream. Note that one leading space but no trailing space or linefeed will be printed. It is okay to pass NULL for A. | ● | ● | (const gcry_mpi_t a) | void |
| gcry_mpi_add | W = U + V. | ● | ● | (gcry_mpi_t w, gcry_mpi_t u, gcry_mpi_t v); | void |
| gcry_mpi_add_ui | W = U + V. V is an unsigned integer. | ● | ● | (gcry_mpi_t w, gcry_mpi_t u, unsigned long v); | void |
| gcry_mpi_addm | W = U + V mod M. | ● | ● | (gcry_mpi_t w, gcry_mpi_t u, gcry_mpi_t v, gcry_mpi_t m) | void |
| gcry_mpi_sub | W = U - V. | ● | ● | (gcry_mpi_t w, gcry_mpi_t u, gcry_mpi_t v); | void |
| gcry_mpi_sub_ui | W = U - V. V is an unsigned integer. | ● | ● | (gcry_mpi_t w, gcry_mpi_t u, unsigned long v ) | void |
| gcry_mpi_subm | W = U - V mod M | ● | ● | (gcry_mpi_t w, gcry_mpi_t u, gcry_mpi_t v, gcry_mpi_t | void |

| | | | | | |
|---|---|---|---|---|---|
| | | | | m) | |
| gcry_mpi_mul | W = U * V. | ● | ● | (gcry_mpi_t w, gcry_mpi_t u, gcry_mpi_t v) | void |
| gcry_mpi_mul_ui | W = U * V. V is an unsigned integer. | ● | ● | (gcry_mpi_t w, gcry_mpi_t u, unsigned long v ) | void |
| gcry_mpi_mulm | W = U * V mod M. | ● | ● | (gcry_mpi_t w, gcry_mpi_t u, gcry_mpi_t v, gcry_mpi_t m) | void |
| gcry_mpi_mul_2exp | W = U * (2 ^ CNT). | ● | ● | (gcry_mpi_t w, gcry_mpi_t u, unsigned long cnt); | void |
| gcry_mpi_div | Q = DIVIDEND / DIVISOR, R = DIVIDEND % DIVISOR, Q or R may be passed as NULL. ROUND should be negative or 0. | ● | ● | (gcry_mpi_t q, gcry_mpi_t r, gcry_mpi_t dividend, gcry_mpi_t divisor, int round) | void |
| gcry_mpi_mod | R = DIVIDEND % DIVISOR | ● | ● | (gcry_mpi_t r, gcry_mpi_t dividend, gcry_mpi_t divisor) | void |
| gcry_mpi_powm | W = B ^ E mod M. | ● | ● | (gcry_mpi_t w, const gcry_mpi_t b, const gcry_mpi_t e, const gcry_mpi_t m) | void |
| gcry_mpi_gcd | Set G to the greatest common divisor of A and B. Return true if the G is 1. | ● | ● | (gcry_mpi_t g, gcry_mpi_t a, gcry_mpi_t b) | int |
| gcry_mpi_invm | Set X to the multiplicative inverse of A mod M. Return true if the value exists. | ● | ● | (gcry_mpi_t x, gcry_mpi_t a, gcry_mpi_t m) | int |
| gcry_mpi_get_nbits | Return the number of bits required to represent A. | ● | ● | (gcry_mpi_t a) | unsigned int |
| gcry_mpi_test_bit | Return true when bit number N (counting from 0) is set in A. | ● | ● | (gcry_mpi_t a, unsigned int n) | int |
| gcry_mpi_set_bit | Set bit number N in A. | ● | ● | (gcry_mpi_t a, unsigned int n) | void |
| gcry_mpi_clear_bit | Clear bit number N in A. | ● | ● | (gcry_mpi_t a, unsigned int n) | void |
| gcry_mpi_set_highbit | Set bit number N in A and clear all bits greater than N. | ● | ● | (gcry_mpi_t a, unsigned int n) | void |
| gcry_mpi_clear_highbit | Clear bit number N in A and all bits greater than N. | ● | ● | (gcry_mpi_t a, unsigned int n) | void |
| gcry_mpi_rshift | Shift the value of A by N bits to the right and store the result in X. | ● | ● | (gcry_mpi_t x, gcry_mpi_t a, unsigned int n) | void |
| gcry_mpi_lshift | Shift the value of A by N bits to the left and store the result in X. | ● | ● | (gcry_mpi_t x, gcry_mpi_t a, unsigned int n) | void |
| gcry_mpi_set_opaque | Store NBITS of the value P points to in A and mark A as an opaque value. WARNING: Never use an opaque MPI for anything thing else then | ● | ● | (gcry_mpi_t a, void *p, unsigned int nbits) | gcry_mpi_t |

| | | | | | |
|---|---|---|---|---|---|
| | gcry_mpi_release, gcry_mpi_get_opaque. | | | | |
| gcry_mpi_get_opaque | Return a pointer to an opaque value stored in A and return its size in NBITS. Note that the returned pointer is still owned by A and that the function should never be used for an non-opaque MPI. | ● | ● | (gcry_mpi_t a, unsigned int *nbits) | void * |
| gcry_mpi_set_flag | Set the FLAG for the big integer A. Currently only the flag GCRYMPI_FLAG_SECURE is allowed to convert A into an big intger stored in "secure" memory. | ● | ● | (gcry_mpi_t a, enum gcry_mpi_flag flag) | void |
| gcry_mpi_clear_flag | Clear FLAG for the big integer A. Note that this function is currently useless as no flags are allowed. | ● | ● | (gcry_mpi_t a, enum gcry_mpi_flag flag) | void |
| gcry_mpi_get_flag | Return true when the FLAG is set for A. | ● | ● | (gcry_mpi_t a, enum gcry_mpi_flag flag) | int |
| gcry_cipher_open (ENC - See Table 5.3a) | Create a handle for algorithm ALGO to be used in MODE. FLAGS may be given as an bitwise OR of the gcry_cipher_flags values. | ● | ● | (gcry_cipher_hd_t *handle, int algo, int mode, unsigned int flags) | gcry_error_t |
| gcry_cipher_close (ENC - See Table 5.3a) | Close the cioher handle H and release all resource. | ● | ● | (gcry_cipher_hd_t h) | void |
| gcry_cipher_ctl (ENC - See Table 5.3a) | Perform various operations on the cipher object H. | ● | ● | (gcry_cipher_hd_t h, int cmd, void *buffer, size_t buflen) | gcry_error_t |
| gcry_cipher_info (ENC - See Table 5.3a) | Retrieve various information about the cipher object H. | ● | ● | (gcry_cipher_hd_t h, int what, void *buffer, size_t *nbytes) | gcry_error_t |
| gcry_cipher_algo_info (ENC - See Table 5.3a) | Retrieve various information about the cipher algorithm ALGO. | ● | ● | (int algo, int what, void *buffer, size_t *nbytes) | gcry_error_t |
| gcry_cipher_algo_name (ENC - See Table 5.3a) | Map the cipher algorithm whose ID is contained in ALGORITHM to a string representation of the algorithm name. For unknown algorithm IDs this function returns "?". | ● | ● | (int algorithm) | const char * |
| gcry_cipher_map_name (ENC - See Table 5.3a) | Map the algorithm name NAME to an cipher algorithm ID. Return 0 if the algorithm name is not known. | ● | ● | (const char *name) | int |
| gcry_cipher_mode_from_oid (ENC - See Table 5.3a) | Given an ASN.1 object identifier in standard IETF dotted decimal format in STRING, return the | ● | ● | (const char *string) | int |

| | | | | | |
|---|---|---|---|---|---|
| | encryption mode associated with that OID or 0 if not known or applicable. | | | | |
| gcry_cipher_encrypt (ENC - See Table 5.3a) | Encrypt the plaintext of size INLEN in IN using the cipher handle H into the buffer OUT which has an allocated length of OUTSIZE. For most algorithms it is possible to pass NULL for in and 0 for INLEN and do a in-place decryption of the data provided in OUT. | ● | ● | (gcry_cipher_hd_t h, void *out, size_t outsize, const void *in, size_t inlen) | gcry_error_t |
| gcry_cipher_decrypt (ENC - See Table 5.3a) | The counterpart to gcry_cipher_encrypt. | ● | ● | (gcry_cipher_hd_t h, void *out, size_t outsize, const void *in, size_t inlen) | gcry_error_t |
| gcry_cipher_setkey (ENC - See Table 5.3a) | Set KEY of length KEYLEN bytes for the cipher handle HD. | ● | ● | (gcry_cipher_hd_t hd, const void *key, size_t keylen) | gcry_error_t |
| gcry_cipher_setiv (ENC - See Table 5.3a) | Set initialization vector IV of length IVLEN for the cipher handle HD. | ● | ● | (gcry_cipher_hd_t hd, const void *iv, size_t ivlen) | gcry_error_t |
| gcry_cipher_reset (ENC - See Table 5.3a)t | Reset the handle to the state after open. | ● | ● | (gcry_cipher_hd_t h) | gcry_error_t |
| gcry_cipher_sync (ENC - See Table 5.3a) | Perform the OpenPGP sync operation if this is enabled for the cipher handle H. | ● | ● | (gcry_cipher_hd_t h) | gcry_error_t |
| gcry_cipher_cts (ENC - See Table 5.3a) | Enable or disable CTS in future calls to gcry_encrypt(). CBC mode only. | ● | ● | (gcry_cipher_hd_t h, size_t on) | gcry_error_t |
| gcry_cipher_setctr (ENC - See Table 5.3a) | Set counter for CTR mode. (CTR,CTRLEN) must denote a buffer of block size length, or (NULL,0) to set the CTR to the all-zero block. | ● | ● | (gcry_cipher_hd_t hd, const void *ctr, size_t ctrlen) | gpg_error_t |
| gcry_cipher_get_algo_keylen (ENC - See Table 5.3a) | Retrieved the key length in bytes used with algorithm A. | ● | ● | (int algo) | size_t |
| gcry_cipher_get_algo_blklen (ENC - See Table 5.3a) | Retrieve the block length in bytes used with algorithm A. | ● | ● | (int algo) | size_t |
| gcry_cipher_test_algo (ENC - See Table 5.3a) | Return 0 if the algorithm A is available for use. | ● | ● | (int algo) | gcry_error_t |
| gcry_cipher_list (ENC - See Table 5.3a) | Get a list consisting of the IDs of the loaded cipher modules. If LIST is zero, write the number of loaded cipher modules to LIST_LENGTH and return. | ● | ● | (int *list, int *list_length) | gcry_error_t |

| | If LIST is non-zero, the first *LIST_LENGTH algorithm IDs are stored in LIST, which must be of according size. In case there are less cipher modules than *LIST_LENGTH, *LIST_LENGTH is updated to the correct number. | | | | |
|---|---|---|---|---|---|
| gcry_pk_encrypt (DSS – See Table 5.3a) | Encrypt the DATA using the public key PKEY and store the result as a newly created S-expression at RESULT. | ● | ● | (gcry_sexp_t *result, gcry_sexp_t data, gcry_sexp_t pkey) | gcry_error_t |
| gcry_pk_decrypt (DSS – See Table 5.3a) | Decrypt the DATA using the private key SKEY and store the result as a newly created S-expression at RESULT. | ● | ● | (gcry_sexp_t *result, gcry_sexp_t data, gcry_sexp_t skey) | gcry_error_t |
| gcry_pk_sign (DSS – See Table 5.3a) | Sign the DATA using the private key SKEY and store the result as a newly created S-expression at RESULT. | ● | ● | (gcry_sexp_t *result, gcry_sexp_t data, gcry_sexp_t skey) | gcry_error_t |
| gcry_pk_verify (DSS – See Table 5.3a) | Check the signature SIGVAL on DATA using the public key PKEY. | ● | ● | (gcry_sexp_t sigval, gcry_sexp_t data, gcry_sexp_t pkey) | gcry_error_t |
| gcry_pk_testkey (DSS – See Table 5.3a) | Check that private KEY is sane. | ● | ● | (gcry_sexp_t key) | gcry_error_t |
| gcry_pk_genkey (DSS – See Table 5.3a) | Generate a new key pair according to the parameters given in S_PARMS. The new key pair is returned in as an S-expression in R_KEY. | ● | ● | (gcry_sexp_t *r_key, gcry_sexp_t s_parms) | gcry_error_t |
| gcry_pk_ctl (DSS – See Table 5.3a) | Catch all function for miscellaneous operations. | ● | ● | (int cmd, void *buffer, size_t buflen) | gcry_error_t |
| gcry_pk_algo_info (DSS – See Table 5.3a) | Retrieve information about the public key algorithm ALGO. | ● | ● | (int algo, int what, void *buffer, size_t *nbytes) | gcry_error_t |
| gcry_pk_algo_name (DSS – See Table 5.3a) | Map the public key algorithm whose ID is contained in ALGORITHM to a string representation of the algorithm name. For unknown algorithm IDs this functions returns "?". | ● | ● | (int algorithm) | const char * |
| gcry_pk_map_name (DSS – See Table 5.3a) | Map the algorithm NAME to a public key algorithm Id. Return 0 if the algorithm name is not known. | ● | ● | (const char* name) | int |
| gcry_pk_get_nbits (DSS – See Table 5.3a) | Return what is commonly referred as the key length for the given public or private KEY. | ● | ● | (gcry_sexp_t key) | unsigned int |
| gcry_pk_get_keygrip (DSS – See Table 5.3a) | Please note that keygrip is still experimental and should not be used without contacting the author. | ● | ● | (gcry_sexp_t key, unsigned char *array) | unsigned char * |

| | | | | | |
|---|---|---|---|---|---|
| gcry_pk_get_curve (DSS - See Table 5.3a) | Return the name of the curve matching KEY. | ● | ● | (gcry_sexp_t key, int iterator, unsigned int *r_nbits) | const char * |
| gcry_pk_get_param (DSS - See Table 5.3a) | Return an S-expression with the parameters of the named ECC curve NAME. ALGO must be set to an ECC algorithm. | ● | ● | (int algo, const char *name) | gcry_sexp_t |
| gcry_pk_test_algo (DSS - See Table 5.3a) | Return 0 if the public key algorithm A is available for use. | ● | ● | (int algo) | gcry_error_t |
| gcry_pk_list (DSS - See Table 5.3a) | Get a list consisting of the IDs of the loaded pubkey modules. If LIST is zero, write the number of loaded pubkey modules to LIST_LENGTH and return. If LIST is non-zero, the first *LIST_LENGTH algorithm IDs are stored in LIST, which must be of according size. In case there are less pubkey modules than *LIST_LENGTH, *LIST_LENGTH is updated to the correct number. | ● | ● | (int *list, int *list_length) | gcry_error_t |
| gcry_md_open (SHS - See Table 5.3a) | Create a message digest object for algorithm ALGO. FLAGS may be given as an bitwise OR of the gcry_md_flags values. ALGO may be given as 0 if the algorithms to be used are later set using gcry_md_enable. | ● | ● | (gcry_md_hd_t *h, int algo, unsigned int flags) | gcry_error_t |
| gcry_md_close (SHS - See Table 5.3a) | Release the message digest object HD. | ● | ● | (gcry_md_hd_t hd) | void |
| gcry_md_enable (SHS - See Table 5.3a) | Add the message digest algorithm ALGO to the digest object HD. | ● | ● | (gcry_md_hd_t hd, int algo) | gcry_error_t |
| gcry_md_copy (SHS - See Table 5.3a) | Create a new digest object as an exact copy of the object HD. | ● | ● | (gcry_md_hd_t *bhd, gcry_md_hd_t ahd) | gcry_error_t |
| gcry_md_reset (SHS - See Table 5.3a) | Reset the digest object HD to its initial state. | ● | ● | (gcry_md_hd_t hd) | void |
| gcry_md_ctl (SHS - See Table 5.3a) | Perform various operations on the digest object HD. | ● | ● | (gcry_md_hd_t hd, int cmd, void *buffer, size_t buflen) | gcry_error_t |
| gcry_md_write (SHS - See Table 5.3a) | Pass LENGTH bytes of data in BUFFER to the digest object HD so that it can update the digest values. This is the actual hash function. | ● | ● | (gcry_md_hd_t hd, const void *buffer, size_t length) | void |

| | | | | | |
|---|---|---|---|---|---|
| gcry_md_read (SHS – See Table 5.3a) | Read out the final digest from HD return the digest value for algorithm ALGO. | ● | ● | (gcry_md_hd_t hd, int algo) | unsigned char * |
| gcry_md_hash_buffer (SHS – See Table 5.3a) | Convenience function to calculate the hash from the data in BUFFER of size LENGTH using the algorithm ALGO avoiding the creating of a hash object. The hash is returned in the caller provided buffer DIGEST which must be large enough to hold the digest of the given algorithm. | ● | ● | (int algo, void *digest, const void *buffer, size_t length) | void |
| gcry_md_get_algo (SHS – See Table 5.3a) | Retrieve the algorithm used with HD. This does not work reliable if more than one algorithm is enabled in HD. | ● | ● | (gcry_md_hd_t hd) | int |
| gcry_md_get_algo_dlen (SHS – See Table 5.3a) | Retrieve the length in bytes of the digest yielded by algorithm ALGO. | ● | ● | (int algo) | unsigned int |
| gcry_md_is_enabled (SHS – See Table 5.3a) | Return true if the the algorithm ALGO is enabled in the digest object A. | ● | ● | (gcry_md_hd_t a, int algo) | int |
| gcry_md_is_secure (SHS – See Table 5.3a) | Return true if the digest object A is allocated in "secure" memory. | ● | ● | (gcry_md_hd_t a) | int |
| gcry_md_info (SHS – See Table 5.3a) | Retrieve various information about the object H. | ● | ● | (gcry_md_hd_t h, int what, void *buffer, size_t *nbytes) | gcry_error_t |
| gcry_md_algo_info (SHS – See Table 5.3a) | Retrieve various information about the algorithm ALGO. | ● | ● | (int algo, int what, void *buffer, size_t *nbytes) | gcry_error_t |
| gcry_md_algo_name (SHS – See Table 5.3a) | Map the digest algorithm id ALGO to a string representation of the algorithm name. For unknown algorithms this function returns "?". | ● | ● | (int algo) | const char * |
| gcry_md_map_name (SHS – See Table 5.3a) | Map the algorithm NAME to a digest algorithm Id. Return 0 if the algorithm name is not known. | ● | ● | (const char* name) | int |
| gcry_md_setkey (SHS – See Table 5.3a) | For use with the HMAC feature, the set MAC key to the KEY of KEYLEN bytes. | ● | ● | (gcry_md_hd_t hd, const void *key, size_t keylen) | gcry_error_t |
| gcry_md_debug (SHS – See Table 5.3a) | Start or stop debugging for digest handle HD; i.e. create a file named dbgmd-<n>.<suffix> while hashing. If SUFFIX is NULL, debugging stops and the file will be closed. | ● | ● | (gcry_md_hd_t hd, const char *suffix) | void |

| | | | | | |
|---|---|---|---|---|---|
| gcry_md_putc | Update the hash(s) of H with the character C. This is a buffered version of the gcry_md_write function. | ● | ● | (gcry_md_hd_t h, char c) | void |
| gcry_md_final (SHS – See Table 5.3a) | Finalize the digest calculation. This is not really needed because gcry_md_read() does this implicitly. | ● | ● | (int algo) | gcry_error_t |
| gcry_md_test_algo (SHS – See Table 5.3a) | Return 0 if the algorithm A is available for use. | ● | ● | (int algo) | gcry_error_t |
| gcry_md_get_asnoid (SHS – See Table 5.3a) | Return an DER encoded ASN.1 OID for the algorithm A in buffer B. N must point to size_t variable with the available size of buffer B. After return it will receive the actual size of the returned OID. | ● | ● | (int algo, void *buffer, size_t *nbytes) | gcry_error_t |
| gcry_md_start_debug (SHS – See Table 5.3a) | Enable debugging for digest object A; i.e. create files named dbgmd-<n>.<string> while hashing. B is a string used as the suffix for the filename. This macro is deprecated, use gcry_md_debug. | ● | ● | (int algo, void *string) | gcry_error_t |
| gcry_md_stop_debug (SHS – See Table 5.3a) | Disable the debugging of A. This macro is deprecated, use gcry_md_debug. | ● | ● | (int algo, void *string) | gcry_error_t |
| gcry_md_list (SHS – See Table 5.3a) | Get a list consisting of the IDs of the loaded message digest modules. If LIST is zero, write the number of loaded message digest modules to LIST_LENGTH and return. If LIST is non-zero, the first *LIST_LENGTH algorithm IDs are stored in LIST, which must be of according size. In case there are less message digest modules than *LIST_LENGTH, *LIST_LENGTH is updated to the correct number. | ● | ● | (int *list, int *list_length) | gcry_error_t |
| gcry_ac_data_new (DSS – See Table 5.3a) | Returns a new, empty data set in DATA. | ● | ● | (gcry_ac_data_t *data) | gcry_error_t |
| gcry_ac_data_destroy (DSS – See Table 5.3a) | Destroy the data set DATA. | ● | ● | (gcry_ac_data_t data) | void |

| | | | | | |
|---|---|---|---|---|---|
| gcry_ac_data_copy (DSS – See Table 5.3a) | Create a copy of the data set DATA and store it in DATA_CP. | ● | ● | (gcry_ac_data_t *data_cp, gcry_ac_data_t data) | gcry_error_t |
| gcry_ac_data_length (DSS – See Table 5.3a) | Return the number of named MPI values inside of the data set DATA. | ● | ● | (gcry_ac_data_t data) | unsigned int |
| gcry_ac_data_clear (DSS – See Table 5.3a) | Destroy any values contained in the data set DATA. | ● | ● | (gcry_ac_data_t data) | void |
| gcry_ac_data_set (DSS – See Table 5.3a)t | Add the value MPI to DATA with the label NAME. If FLAGS contains GCRY_AC_FLAG_DATA_COPY, the data set will contain copies of NAME and MPI. If FLAGS contains GCRY_AC_FLAG_DATA_DEALLOC or GCRY_AC_FLAG_DATA_COPY, the values contained in the data set will be deallocated when they are to be removed from the data set. | ● | ● | (gcry_ac_data_t data, unsigned int flags, const char *name, gcry_mpi_t mpi) | gcry_error_t |
| gcry_ac_data_get_name (DSS – See Table 5.3a) | Store the value labeled with NAME found in DATA in MPI. If FLAGS contains GCRY_AC_FLAG_COPY, store a copy of the MPI value contained in the data set. MPI may be NULL. | ● | ● | (gcry_ac_data_t data, unsigned int flags, const char *name, gcry_mpi_t *mpi) | gcry_error_t |
| gcry_ac_data_get_index (DSS – See Table 5.3a) | Stores in NAME and MPI the named MPI value contained in the data set DATA with the index IDX. If FLAGS contains GCRY_AC_FLAG_COPY, store copies of the values contained in the data set. NAME or MPI may be NULL. | ● | ● | (gcry_ac_data_t data, unsigned int flags, unsigned int idx, const char **name, gcry_mpi_t *mpi) | gcry_error_t |
| gcry_ac_data_to_sexp (DSS – See Table 5.3a) | Convert the data set DATA into a new S-Expression, which is to be stored in SEXP, according to the identifiers contained in IDENTIFIERS. | ● | ● | (gcry_ac_data_t data, gcry_sexp_t *sexp, const char **identifiers) | gcry_error_t |
| gcry_ac_data_from_sexp (DSS – See Table 5.3a) | Create a new data set, which is to be stored in DATA_SET, from the S-Expression SEXP, according to the identifiers contained in IDENTIFIERS. | ● | ● | (gcry_ac_data_t *data, gcry_sexp_t sexp, const char **identifiers) | gcry_error_t |
| gcry_ac_io_init (DSS – See Table | Initialize AC_IO according to MODE, TYPE and the | ● | ● | (gcry_ac_io_t *ac_io, gcry_ac_io_mode_t | void |

| | | | | | |
|---|---|---|---|---|---|
| 5.3a) | variable list of arguments. The list of variable arguments to specify depends on the given TYPE. | | | mode, gcry_ac_io_type_t type, ...) | |
| gcry_ac_io_init_va (DSS – See Table 5.3a) | Initialize AC_IO according to MODE, TYPE and the variable list of arguments AP. The list of variable arguments to specify depends on the given TYPE. | ● | ● | (gcry_ac_io_t *ac_io, gcry_ac_io_mode_t mode, gcry_ac_io_type_t type, va_list ap) | void |
| gcry_ac_open (DSS – See Table 5.3a) | Create a new ac handle. | ● | ● | (gcry_ac_handle_t *handle, gcry_ac_id_t algorithm, unsigned int flags) | gcry_error_t |
| gcry_ac_close (DSS – See Table 5.3a) | Destroy an ac handle. | ● | ● | (gcry_ac_handle_t handle) | void |
| gcry_ac_key_init (DSS – See Table 5.3a) | Initialize a key from a given data set. | ● | ● | (gcry_ac_key_t *key, gcry_ac_handle_t handle, gcry_ac_key_type_t type, gcry_ac_data_t data) | gcry_error_t |
| gcry_ac_key_pair_generate (DSS – See Table 5.3a) | Generates a new key pair via the handle HANDLE of NBITS bits and stores it in KEY_PAIR. In case non-standard settings are wanted, a pointer to a structure of type gcry_ac_key_spec_<algorithm>_t, matching the selected algorithm, can be given as KEY_SPEC. MISC_DATA is not used yet. | ● | ● | (gcry_ac_handle_t handle, unsigned int nbits, void *spec, gcry_ac_key_pair_t *key_pair, gcry_mpi_t **misc_data) | gcry_error_t |
| gcry_ac_key_pair_extract (DSS – See Table 5.3a) | Returns the key of type WHICH out of the key pair KEY_PAIR. | ● | ● | (gcry_ac_key_pair_t key_pair, gcry_ac_key_type_t which) | gcry_ac_key_t |
| gcry_ac_key_data_get (DSS – See Table 5.3a) | Returns the data set contained in the key KEY. | ● | ● | (gcry_ac_key_t key) | gcry_ac_data_t |
| gcry_ac_key_test (DSS – See Table 5.3a)t | Verifies that the key KEY is sane via HANDLE. | ● | ● | (gcry_ac_handle_t handle, gcry_ac_key_t key) | gcry_error_t |
| gcry_ac_key_get_nbits (DSS – See Table 5.3a) | Stores the number of bits of the key KEY in NBITS via HANDLE. | ● | ● | (gcry_ac_handle_t handle, gcry_ac_key_t key, unsigned int *nbits) | gcry_error_t |
| gcry_ac_key_get_grip (DSS – See Table 5.3a) | Writes the 20 byte long key grip of the key KEY to KEY_GRIP via HANDLE. | ● | ● | (gcry_ac_handle_t handle, gcry_ac_key_t key, unsigned char *key_grip) | gcry_error_t |
| gcry_ac_key_destroy (DSS – See Table 5.3a) | Destroy a key. | ● | ● | (gcry_ac_key_t key) | void |

| | | | | | |
|---|---|---|---|---|---|
| gcry_ac_key_pair_destroy (DSS – See Table 5.3a) | Destroy a key pair. | ● | ● | (gcry_ac_key_pair_t key_pair) | void |
| gcry_ac_data_encode (DSS – See Table 5.3a) | Encodes a message according to the encoding method METHOD. OPTIONS must be a pointer to a method-specific structure (gcry_ac_em*_t). | ● | ● | (gcry_ac_em_t method, unsigned int flags, void *options, gcry_ac_io_t *io_read, gcry_ac_io_t *io_write) | gcry_error_t |
| gcry_ac_data_decode (DSS – See Table 5.3a) | Decodes a message according to the encoding method METHOD. OPTIONS must be a pointer to a method-specific structure (gcry_ac_em*_t). | ● | ● | (gcry_ac_em_t method, unsigned int flags, void *options, gcry_ac_io_t *io_read, gcry_ac_io_t *io_write) | gcry_error_t |
| gcry_ac_data_encrypt (DSS – See Table 5.3a) | Encrypt the plain text MPI value DATA_PLAIN with the key KEY under the control of the flags FLAGS and store the resulting data set into DATA_ENCRYPTED. | ● | ● | (gcry_ac_handle_t handle, unsigned int flags, gcry_ac_key_t key, gcry_mpi_t data_plain, gcry_ac_data_t *data_encrypted) | gcry_error_t |
| gcry_ac_data_decrypt (DSS – See Table 5.3a) | Decrypt the decrypted data contained in the data set DATA_ENCRYPTED with the key KEY under the control of the flags FLAGS and store the resulting plain text MPI value in DATA_PLAIN. | ● | ● | (gcry_ac_handle_t handle, unsigned int flags, gcry_ac_key_t key, gcry_mpi_t *data_plain, gcry_ac_data_t data_encrypted) | gcry_error_t |
| gcry_ac_data_sign (DSS – See Table 5.3a) | Sign the data contained in DATA with the key KEY and store the resulting signature in the data set DATA_SIGNATURE. | ● | ● | (gcry_ac_handle_t handle, gcry_ac_key_t key, gcry_mpi_t data, gcry_ac_data_t *data_signature) | gcry_error_t |
| gcry_ac_data_verify (DSS – See Table 5.3a) | Verify that the signature contained in the data set DATA_SIGNATURE is indeed the result of signing the data contained in DATA with the secret key belonging to the public key KEY. | ● | ● | (gcry_ac_handle_t handle, gcry_ac_key_t key, gcry_mpi_t data, gcry_ac_data_t data_signature) | gcry_error_t |
| gcry_ac_data_encrypt_scheme (DSS – See Table 5.3a) | Encrypts the plain text readable from IO_MESSAGE through HANDLE with the public key KEY according to SCHEME, FLAGS and OPTS. If OPTS is not NULL, it has to be a pointer to a structure specific to the chosen scheme (gcry_ac_es_*_t). The encrypted message is written to IO_CIPHER. | ● | ● | (gcry_ac_handle_t handle, gcry_ac_scheme_t scheme, unsigned int flags, void *opts, gcry_ac_key_t key, gcry_ac_io_t *io_message, gcry_ac_io_t *io_cipher) | gcry_error_t |
| gcry_ac_data_decrypt_scheme | Decrypts the cipher text readable from IO_CIPHER | ● | ● | (gcry_ac_handle_t handle, | gcry_error_t |

| | | | | | |
|---|---|---|---|---|---|
| (DSS - See Table 5.3a) | through HANDLE with the secret key KEY according to SCHEME, @var{flags} and OPTS. If OPTS is not NULL, it has to be a pointer to a structure specific to the chosen scheme (gcry_ac_es_*_t). The decrypted message is written to IO_MESSAGE. | | | gcry_ac_scheme_t scheme, unsigned int flags, void *opts, gcry_ac_key_t key, gcry_ac_io_t *io_cipher, gcry_ac_io_t *io_message) | |
| gcry_ac_data_sign_scheme (DSS - See Table 5.3a) | Signs the message readable from IO_MESSAGE through HANDLE with the secret key KEY according to SCHEME, FLAGS and OPTS. If OPTS is not NULL, it has to be a pointer to a structure specific to the chosen scheme (gcry_ac_ssa_*_t). The signature is written to IO_SIGNATURE. | ● | ● | (gcry_ac_handle_t handle, gcry_ac_scheme_t scheme, unsigned int flags, void *opts, gcry_ac_key_t key, gcry_ac_io_t *io_message, gcry_ac_io_t *io_signature) | gcry_error_t |
| gcry_ac_data_verify_scheme (DSS - See Table 5.3a) | Verifies through HANDLE that the signature readable from IO_SIGNATURE is indeed the result of signing the message readable from IO_MESSAGE with the secret key belonging to the public key KEY according to SCHEME and OPTS. If OPTS is not NULL, it has to be an anonymous structure (gcry_ac_ssa_*_t) specific to the chosen scheme. | ● | ● | (gcry_ac_handle_t handle, gcry_ac_scheme_t scheme, unsigned int flags, void *opts, gcry_ac_key_t key, gcry_ac_io_t *io_message, gcry_ac_io_t *io_signature) | gcry_error_t |
| gcry_ac_id_to_name (DSS - See Table 5.3a) | Store the textual representation of the algorithm whose id is given in ALGORITHM in NAME. This function is deprecated; use gcry_pk_algo_name. | ● | ● | (gcry_ac_id_t algorithm, const char **name) | gcry_error_t |
| gcry_ac_name_to_id (DSS - See Table 5.3a) | Store the numeric ID of the algorithm whose textual representation is contained in NAME in ALGORITHM. This function is deprecated; use gcry_pk_map_name. | ● | ● | (const char *name, gcry_ac_id_t *algorithm) | gcry_error_t |
| gcry_kdf_derive (KDF – See Table 5.3a) | Derive a key from a passphrase. | ● | ● | (const void *passphrase, size_t passphraselen, int algo, int subalgo, const void *salt, size_t saltlen, unsigned long iterations, size_t keysize, void *keybuffer) | gpg_error_t |
| gcry_randomize (RNG – See Table 5.3a) | Fill BUFFER with LENGTH bytes of random, using random numbers of quality LEVEL. | ● | ● | (void *buffer, size_t length, enum gcry_random_level level) | void |

| | | | | | |
|---|---|---|---|---|---|
| gcry_random_add_bytes (RNG – See Table 5.3a) | Add the external random from BUFFER with LENGTH bytes into the pool. QUALITY should either be -1 for unknown or in the range of 0 to 100 | ● | ● | (const void *buffer, size_t length, int quality) | gcry_error_t |
| gcry_fast_random_poll (RNG – See Table 5.3a) | If random numbers are used in an application, this macro should be called from time to time so that new stuff gets added to the internal pool of the RNG. | ● | ● | None | gcry_error_t |
| gcry_random_bytes (RNG – See Table 5.3a) | Return NBYTES of allocated random using a random numbers of quality LEVEL. | ● | ● | (size_t nbytes, enum gcry_random_level level) | void * |
| gcry_random_bytes_secure (RNG – See Table 5.3a) | Return NBYTES of allocated random using a random numbers of quality LEVEL. The random numbers are created returned in "secure" memory. | ● | ● | (size_t nbytes, enum gcry_random_level level) | void * |
| gcry_mpi_randomize | Set the big integer W to a random value of NBITS using a random generator with quality LEVEL. Note that by using a level of GCRY_WEAK_RANDOM gcry_create_nonce is used internally. | ● | ● | (gcry_mpi_t w, unsigned int nbits, enum gcry_random_level level) | void |
| gcry_create_nonce (RNG – See Table 5.3a) | Create an unpredictable nonce of LENGTH bytes in BUFFER. | ● | ● | (void *buffer, size_t length) | void |
| gcry_prime_generate | Generate a new prime number of PRIME_BITS bits and store it in PRIME. If FACTOR_BITS is non-zero, one of the prime factors of (prime - 1) / 2 must be FACTOR_BITS bits long. If FACTORS is non-zero, allocate a new, NULL-terminated array holding the prime factors and store it in FACTORS. FLAGS might be used to influence the prime number generation process. | ● | ● | (gcry_mpi_t *prime, unsigned int prime_bits, unsigned int factor_bits, gcry_mpi_t **factors, gcry_prime_check_func_t cb_func, void *cb_arg, gcry_random_level_t random_level, unsigned int flags) | gcry_error_t |
| gcry_prime_group_generator | Find a generator for PRIME where the factorization of (prime-1) is in the NULL terminated array FACTORS. Return the generator as a newly allocated MPI in R_G. If START_G is not NULL, use this as the start for the search. | ● | ● | (gcry_mpi_t *r_g, gcry_mpi_t prime, gcry_mpi_t *factors, gcry_mpi_t start_g) | gcry_error_t |

| | | | | | |
|---|---|---|---|---|---|
| gcry_prime_release_factors | Convenience function to release the FACTORS array. | ● | ● | (gcry_mpi_t *factors) | void |
| gcry_prime_check | Check whether the number X is prime. | ● | ● | (gcry_mpi_t x, unsigned int flags) | gcry_error_t |
| gcry_set_progress_handler | Certain operations can provide progress information. This function is used to register a handler for retrieving these information. | ● | ● | (gcry_handler_progress_t cb, void *cb_data) | void |
| gcry_set_allocation_handler | Register a custom memory allocation functions. | ● | ● | ( gcry_handler_alloc_t func_alloc, gcry_handler_alloc_t func_alloc_secure, gcry_handler_secure_check_t func_secure_check, gcry_handler_realloc_t func_realloc, gcry_handler_free_t func_free) | void |
| gcry_set_outofcore_handler | Register a function used instead of the internal out of memory handler. | ● | ● | (gcry_handler_no_mem_t h, void *opaque) | void |
| gcry_set_fatalerror_handler | Register a function used instead of the internal fatal error handler. | ● | ● | (gcry_handler_error_t fnc, void *opaque) | void |
| gcry_set_log_handler | Register a function used instead of the internal logging facility. | ● | ● | (gcry_handler_log_t f, void *opaque) | void |
| gcry_set_gettext_handler | Reserved for future use. | ● | ● | (const char *(*f)(const char*)) | void |
| gcry_malloc | Libgcrypt uses its own memory allocation. It is important to use gcry_free () to release memory allocated by libgcrypt. | ● | ● | (size_t n) | void * |
| gcry_calloc | | | | (size_t n, size_t m) | void * |
| gcry_malloc_secure | | | | (size_t n) | void * |
| gcry_calloc_secure | | | | (size_t n, size_t m) | void * |
| gcry_realloc | | | | (void *a, size_t n) | void * |
| gcry_strdup | | | | (const char *string) | char * |
| gcry_xmalloc | | | | (size_t n) | void * |
| gcry_xcalloc | | | | (size_t n, size_t m) | void * |
| gcry_xmalloc_secure | | | | (size_t n) | void * |
| gcry_xcalloc_secure | | | | (size_t n, size_t m) | void * |
| gcry_xrealloc | | | | (void *a, size_t n) | void * |
| gcry_xstrdup | | | | (const char * a) | char * |
| gcry_free | | | | (void *a) | void |
| gcry_is_secure | Return true if A is allocated in "secure" memory. | ● | ● | (const void *a) | int |
| gcry_fips_mode_active | Return true if Libgcrypt is in FIPS mode. | ● | ● | None | bool |

**Table 5.3a Libgcrypt Algorithms available in non-Approved mode**

| Libgcrypt Service Group | Available Algorithms |
|---|---|
| **ENC** (Encryption) | AES128, non-compliant<br>AES192, non-compliant<br>AES256, non-compliant<br>ARCFOUR<br>BLOWFISH<br>CAMELLIA128<br>CAMELLIA192<br>CAMELLIA256<br>CAST5<br>DES<br>RC2<br>SEED<br>SERPENT128<br>SERPENT192<br>SERPENT256<br>TDES<br>TWOFISH<br>TWOFISH128 |
| **DSS** (Digital Signature Standard) | RSA, non-compliant<br>DSA, non-compliant<br>Elgamal |
| **SHS** (Secure Hash Standard) | If flag GCRY_MD_FLAG_HMAC is 0, then the functions perform the specified hash algorithm<br>If flag GCRY_MD_FLAG_HMAC is 1, then the functions perform the specified HMAC algorithm<br>HAVAL<br>MD2<br>MD4<br>MD5<br>RMD160<br>SHA1, non-compliant<br>SHA224, non-compliant<br>SHA256, non-compliant<br>SHA384, non-compliant<br>SHA512, non-compliant<br>TIGER<br>TIGER1<br>TIGER2<br>WHIRLPOOL |
| **RNG** (Random Number Generators) | NIST-Recommended Random Number Generator Based on ANSI X9.31 Appendix A.2.4 Using the 3-Key Triple DES and AES Algorithms (Implementing AES Only), non-compliant<br>CSPRNG - The Continuously Seeded Pseudo Random Number Generator |
| **KDF** (Key Derivation Function) | SIMPLE_S2K - The OpenPGP simple S2K algorithm (cf. RFC4880). Its use is strongly deprecated.<br>SALTED_S2K - The OpenPGP salted S2K algorithm (cf. RFC4880). Usually not used.<br>ITERSALTED_S2K - The OpenPGP iterated+salted S2K algorithm (cf. RFC4880). This is the default for most OpenPGP applications.<br>PBKDF2 - The PKCS#5 Passphrase Based Key |

| | Derivation Function number 2. |
|---|---|

### 3.3 Authentication

The Module neither identifies nor authenticates any user (in any role) that is accessing the Module. This is only acceptable for a FIPS 140-2, Security Level 1 validation. The Operating System (OS) and the SecureAgent Application provide functionality to require any user to be successfully authenticated prior to using any system services. Only a single operator assuming a particular role may operate the Module at any particular moment in time. The OS authentication mechanism must be enabled to ensure that none of the Module's services are available to users who do not assume an authorized role.

***Table 6. Roles and Required Identification and Authentication.***

| ***Role*** | ***Type of Authentication*** | ***Authentication Data*** |
|---|---|---|
| **Crypto Officer** | Not required | Not required |
| **User** | Not required | Not required |

## 4 Finite State Model

Refer to ***SecureAgent FiniteStateMachine*** for information related to the finite state model implemented in this module.

## 5 Physical Security (N/A)

This is a software module.

## 6 Operational Environment

The Module must run on Sun Solaris 10.

After the module is initialized, a crypto officer confirms the module has the correct version number using the sai_version function.

2.2.006

## 7 Cryptographic Key Management

SecureAgent® Software Cryptographic Module performs no key management. Because the SecureAgent® Software Cryptographic Module is a DLL, each process requesting access is provided its own instance of the SecureAgent® Software Cryptographic Module. The SecureAgent® Software Cryptographic Module contains only keys or data placed into the SecureAgent® Software Cryptographic Module via the services described in this document. No keys or data are persistently maintained by the SecureAgent® Software Cryptographic Module, or maintained after a process detaches from the SecureAgent® Software Cryptographic Module.

***Table 7. Keys and CSPs***

| ***Name*** | ***Size*** | ***Description*** | ***Method*** | ***CSP Access*** |
|---|---|---|---|---|
| Encrypt-Key | 128,192,256 for AES | Key supplied to the Send (encrypt data) service | API call | Crypto Officer, User |
| Decrypt-Key | 128,192,256 for AES | Key supplied to the Receive (decrypt data) service | API call | Crypto Officer, User |
| Seed | 128 | Seed generated from NDRNG used to seed X9.31 RNG | API call | Crypto Officer, User |
| Seed Key | 128 | Seed key generated from NDRNG used as seed key for X9.31 RNG | API call | Crypto Officer, User |

### 7.1 Random Number Generators

The SecureAgent® Software Cryptographic Module uses an Approved RNG (ANSI X9.31 Appendix A.2.4 using AES) to generate initialization vectors (IVs) for Approved security functions.
The SecureAgent® Software Cryptographic Module uses a non-Approved NDRNG (Sun Solaris 10 /dev/random) to seed the above Approved RNG.

### 7.2 Key Generation

The SecureAgent® Software Cryptographic Module does not provide key generation services. Keys are generated outside the module. All keys must be provided through the API as described.

### 7.3 Key Establishment

No key establishment is performed by the module.

### 7.4 Key Entry and Output

Plaintext keys are entered into the module as parameters to API calls. No manual key entry is possible with the module.

### 7.5 Key Storage

The SecureAgent® Software Cryptographic Module does not provide any persistent storage of key material. Keys are entered by the operator only via API calls. Key material is stored in the context, which is maintained in a user-supplied data structure passed in each API call. No key material is maintained inside the SecureAgent® Software Cryptographic Module between API calls. The only key material used by the SecureAgent® Software Cryptographic Module outside of the user-supplied context is that which is stored temporarily in local variables on the stack.

The SecureAgent® Software Cryptographic Module relies upon the operating system memory protection to prevent processes from accessing each other's key material. To ensure that other processes cannot access keys and data, the caller must not utilize shared memory. In addition, the operating system page file must not be configured to reside on a network drive.

### 7.6 Key Zeroization

Zeroization of keys is performed internal to the module API functions using the gcry_burn_stack function and the wipememory macro. sai_zeroize reimplements the wipememory macro for application level key/CSP zeroization.

It is possible that the operating system may swap memory that contains keys to a disk. To zeroize those keys, the User must wipe the swap files. One way to accomplish this is to reformat the hard drive containing the swap filesystem and overwrite the hard drive at least once.

## 8 Electromagnetic Interference/Electromagnetic Compatibility (EMI/EMC)

The module runs on a GPC and this will be required to comply with FCC Part 15 regulations regarding the EMI/EMC for business use.

## 9 Self-Tests

The module implements several self-tests to check proper functioning of the module. This includes power-up self-tests (which are also callable on demand) and conditional self-tests. The self-test can be initiated by calling the function sai_selftest, which returns the operational status of the module (after running self-tests) and an error code with description of the error (if applicable). Additionally, when the module is performing self-tests, none of the FIPS approved algorithms will be available and no data output is possible until self-tests are successfully completed.

### 9.1 Power-Up Tests

The module performs a self-test when the API function sai_init is called or on demand when the API function sai_selftest is called. When the module is initialized with sai_init, a Known Answer Test is performed for the each cryptographic algorithm. The module returns the operational status of the module (after running self-tests) as “Pass/Fail” as shown in Table 5.1. If **any** of these tests fail, the module enters the fatal error state. Once the module is in the fatal error state, none of the FIPS approved algorithms will be available and data output is inhibited. Both functions may be called anytime.

**9.1.1** Cryptographic algorithm tests:

- AES 128, 192, 256
- HMAC-SHA256
- SHA256
- RNG

**9.1.2** Software integrity check uses an HMAC-SHA256 function to compare calculated result with a previously generated result to verify module integrity.

### *9.2 Conditional Tests*

**9.2.1** Software integrity check uses an HMAC-SHA256 function to compare calculated result with a previously generated result to verify module integrity.

**9.2.2** Approved RNG continuous random number generator test checks for failure to a constant value.

**9.2.3** Non-Approved NDRNG continuous random number generator test checks for failure to a constant value.

## 10 Design Assurance

### *10.1 Configuration Management*

A configuration management (CM) system is established for the cryptographic module, module components, and associated module documentation. The configuration management defines how version numbers correspond to changes made to configuration items during the product lifecycle. The vendor uses the Mercurial source control management tool to facilitate the integrity of the module during the module's life cycle. Mercurial stores all changes to the Module's source code.

### *10.2 Delivery and Operation*

The SecureAgent® Software Cryptographic Module is provided as a dynamically-linked shared object library for the Solaris operating environment.

### *10.3 Development*

The cryptographic module software consists of the runtime software. Listings of these sources are provided separately. Annotations in the source code modules explain functions implemented in the code.

### *10.4 Guidance Documents*

The Crypto Officer Guide provides instructions on how to properly install and configure the Module. The User Guide describes how to use the services available in the Module.

## 11 Mitigation of Attacks (N/A)

The SecureAgent® Software cryptographic module is not designed to mitigate specific attacks such as differential power analysis or timing attacks.

## References

National Institute of Standards and Technology, *FIPS PUB 140-2: Security Requirements for Cryptographic Modules*, available at URL: http://csrc.nist.gov/groups/STM/cmvp.

National Institute of Standards and Technology, *FIPS 140-2 Annex A: Approved Security Functions*, available at URL: http://csrc.nist.gov/groups/STM/cmvp.

National Institute of Standards and Technology, *FIPS 140-2 Annex B: Approved Protection Profiles*, available at URL: http://csrc.nist.gov/groups/STM/cmvp.

National Institute of Standards and Technology, *FIPS 140-2 Annex C: Approved Random Number Generators*, available at URL: http://csrc.nist.gov/groups/STM/cmvp.

National Institute of Standards and Technology, *FIPS 140-2 Annex D: Approved Key Establishment Techniques*, available at URL: http://csrc.nist.gov/groups/STM/cmvp.

National Institute of Standards and Technology and Communications Security Establishment, *Derived Test Requirements (DTR) for FIPS PUB 140-2, Security Requirements for Cryptographic Modules*, available at URL: http://csrc.nist.gov/groups/STM/cmvp.

National Institute of Standards and Technology, *Data Encryption Standard (DES)*, Federal Information Processing Standards Publication 46-3, available at URL: http://csrc.nist.gov/groups/STM/cmvp.

National Institute of Standards and Technology, *DES Modes of Operation*, Federal Information Processing Standards Publication 81, available at URL: http://csrc.nist.gov/groups/STM/cmvp.

National Institute of Standards and Technology, *Digital Signature Standard (DSS)*, Federal Information Processing Standards Publication 186-2, available at URL: http://csrc.nist.gov/groups/STM/cmvp.

National Institute of Standards and Technology, *Secure Hash Standard (SHS)*, Federal Information Processing Standards Publication 180-1, available at URL: http://csrc.nist.gov/groups/STM/cmvp.