SecureCore

SecureCore Prototype/Demo Manual:
Definition & Concept of Operations,
SP TSM Interfaces,
TML & SCOS Interface,
and SP Emulation Module Interface

Princeton University & ISI

Version 1.1 – 8/25/2009
Jeffrey Dwoskin, Ganesha Bhaskara, Thuy D. Nguyen and Ruby Lee

DRAFT
For internal discussion only

(This page intentionally left blank)

Table of Contents **Page**

I. Introduction

This document describes the architecture and implementation details for the SecureCore prototype/demo [SCDemo]. The prototype will be a simplified software implementation of the real SecureCore architecture, as shown in Figure 1.

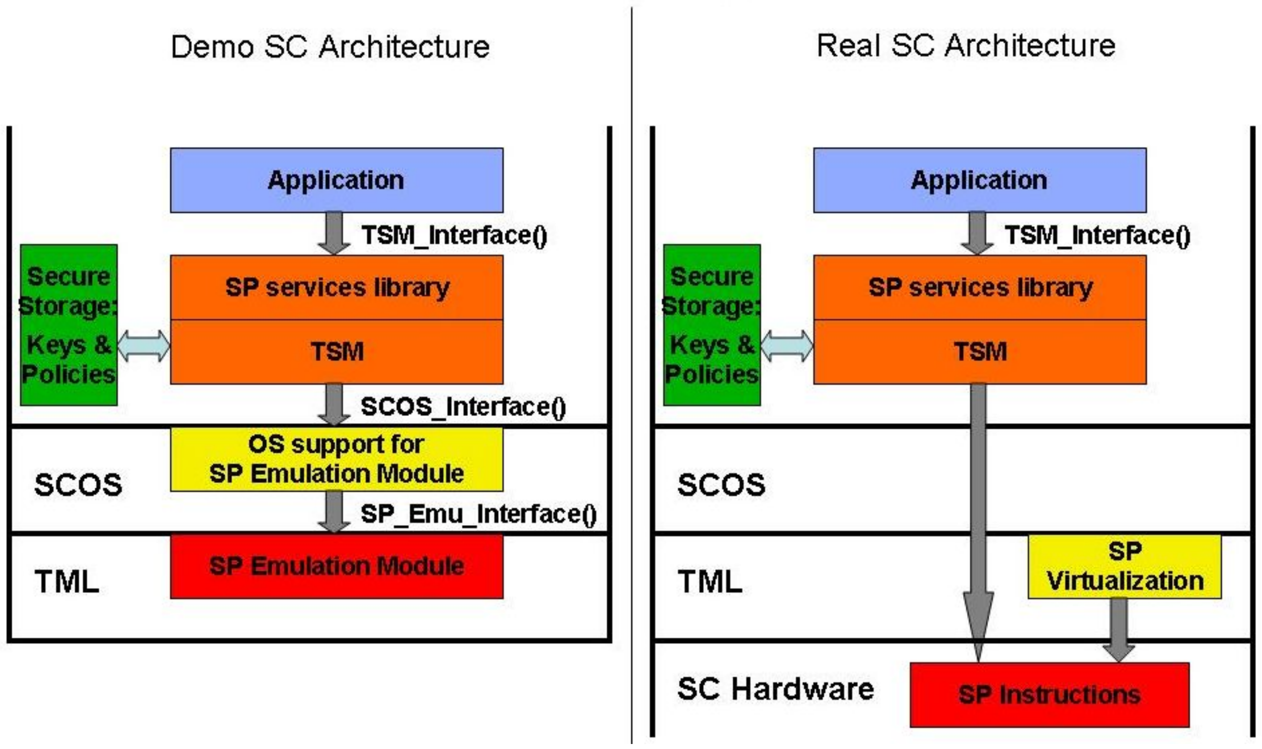

Figure 1: SecureCore Architecture

The SP hardware features will be implemented as a software emulation module running inside the TML. The SP Emulation Module API in Section V documents its interface, including both requirements and interfaces for the TML and interfaces provided to user-level applications. These user-level interfaces are exported by the SP Emulation Module to the SCOS, which in turn exports them to applications where they are used by TSM code. The TSM implements its own interfaces, specified in Section III, and provides an abstraction of authority- and third-party-managed keychains with associated policies and access control. The TSM is called via the SP Services Library (using internal calling conventions), which provides the services to a key-management application. The semantics and internal structures of the TSM are described in Section II, and define the operations of the key-management application.

II. Definition of Key-management Application

a) Introduction

This section describes the purpose and design of the key management application and its interaction with the SP TSM. The TSM will provide (and protect) multiple keychains on behalf of the authority and any authorized third parties. Each keychain can contain multiple keys belonging to either the authority or a single third party. The keys are associated with key usage policies. The keys are then used on the SecureCore device through the key management application which makes all accesses through the TSM. The TSM will allow the keys to be used only according to the policies, and will never reveal the keys themselves to other software.

The authority is able to add/delete keychains from the device by managing a set of encryption/MAC keys that is unique for each keychain. Those keys can then be used to add/delete keys within each keychain, either by the authority itself or by a third party that is given the keys. There is no fixed limit to the number of keychains or the number of keys per keychain.

b) Assumptions & Definitions

<u>Assumptions:</u>

- All keys used by the TSM (for encryption and signing) are symmetric keys.
- The authority and each third party securely exchange, out of band, a unique pair of symmetric keys which are used for encryption (ENC_KEY) and generating a MAC (MAC_KEY) respectively. These are used for sending authenticated command messages to the device. (If a single third party controls multiple keychains, it can have multiple pairs of keys.)
- The authority assigns and communicates to each third party a globally unique identifier we call key chain identifier (KC_ID) for each keychain.
- The assurance of enforcement of per-user key usage policies by TSM is only as high as the entity supplying the user identification. Without reliable user identification, other users may be able to access keys with the permissions assigned to the primary user of each key.

<u>Definitions:</u>

- Owner
  - An owner of a keychain. The party, either the authority or a third party data provider, who is authorized to manage a particular keychain and can add or remove keys from that keychain and set the policy for those keys.
- Primary User
  - A user on the SecureCore device, identified and authenticated by the operating system, who is listed as the primary user of a particular key of a keychain. The allowed actions of a policy applied to a key can be different for the Primary User and all Other Users. The TSM enforces this distinction on behalf of the OS or calling application, but does not itself authenticate the user. It trusts the user identification (or possibly an authentication token) provided in the TSM function call.

- Master Keychain
    - A special keychain, with ID = 1, used by the TSM to store the keys used to authenticate messages for updating the other keychains. This master keychain is managed only by the authority. Its keys will be set with policies that deny all actions by the user on the device, as these keys are only used internally by the TSM.
- Command/Update Message
    - A command message or an update message is a signed message sent by either the authority or a third party owner of a keychain to add or delete a key on a keychain, along with its associated policy. The message follows the structure provided below, and is signed with a MAC using the signing key for the keychain that is stored in the master keychain. Messages from the authority to update the master keychain are signed with a MAC using a derived key.
- Keychain Message Counter
    - A monotonically increasing counter kept for each keychain to prevent replay attacks on command/update messages. The authority and third parties keep a similar counter for each keychain they manage and send a new (incremented) count on each command message they send. The device will only accept a message as fresh if the count in the message is greater than the one stored with the keychain. It then updates the stored value to the one in the message.

c) Concept of Operations

.The demonstration of the capabilities of the integrated SP/TML design will consists of a series of serialized operations. Following are the subjects who are referred to in the usage scenarios:

<u>Subjects</u>

- Authority (AUTH)
- Third party A (3P-A)
- Third party B (3P-B)
- User X of the device (User_X)
- User Y of the device (User_Y)
- SecureCore device with unique DRK (DEVICE)

As a first, the user will not be able to select the sequence of individual operations. The demo application will output its progress between steps and wait for user input (key "n") before the next operation is performed. If time permits, we will later provide a menu to exercise the functions interactively. Following is the high level sequence of operations, with additional details in Section **Error! Reference source not found.**:

1. Initialization
    a. SP emulation module in TML
        i. Hardware power-on and lock DRK
    b. Key-management application which includes the TSM and demo front–end
        i. TSM initializes SecureArea and loads Secure Storage
2. Manage Keychains
    a. Authority adds a new keychain for 3P-A
    b. Authority adds a new keychain for 3P-B
    c. Attempt attacks on adding keychains
        i. Replay message for adding 3P-A keychain
        ii. Corrupt/modify an add message in-transit

        iii. 3P-A attempts to add a keychain
3. Manage keys on keychains
    a. 3P-A adds keys: 3P-A-Key1-User_X, 3P-A-Key2-User_Y
    b. 3P-A deletes existing key: 3P-A-KeyB
    c. 3P-B adds keys: 3P-B-Key1, 3P-B-Key2
    d. Attempt attacks on keys
        i. 3P-A attempts to add key to master keychain: Fake-Key1
        ii. 3P-B attempts to delete key from 3P-A's keychain: 3P-A-Key1
        iii. Replay and/or modify message for adding key from 3P-B for 3P-B-Key1
        iv. 3P-A modifies a previous message from 3P-B to try to add key 3P-B-Key1 to its own keychain
4. Access control on usage of keys
    a. Login as User_X
    b. Use keys from 3P-A to encrypt/decrypt/sign/MAC/re-encrypt data according to policy.
    c. Demonstrate attacks on access control
        i. Exceed usage limit on a key
        ii. Attempt to perform operations not permitted for User_X
        iii. Attempt to perform operations not permitted for any user
        iv. Attempt to use a key that was deleted
    d. Login as User_Y
    e. Use keys from 3P-A according to policy that were not available to User_X
    f. Demonstrate attacks on access control
        i. Attempt to perform operations not permitted for User_Y that were permitted for User_X
5. Modification of Secure Storage
    a. Save modified keychains in Secure Storage and update SRH
    b. Demonstrate attacks on Secure Storage
        i. Attempt to reload old version of Secure Storage and detect replay attack
6. Demonstrate attacks on memory protection
    a. Attempt to read/modify/replay data in SecureAreas from key-management application, in-between calls to the TSM (after an EndCEM and before the next BeginCEM)
    b. Attempt to read/modify/replay data in SecureAreas during an interrupt
    c. Attempt to read/modify/replay data in saved CEM registers during an interrupt
7. Demonstrate attacks on SP registers
    a. Attempt to reset the DRK after it has been locked
    b. Attempt to use CEM-only instructions from outside of TSM code

Resilience against the following attacks will be demonstrated:

1. Loading/unloading secure storage
    a. Replay attack of encrypted secure storage structure stored on disk
2. Messages to create/delete key chains
    a. Demonstrate attack by corrupted/modified packet
    b. Demonstrate replay attack
    c. Demonstrate attack where 3P tries to add/delete a keychain
3. Messages to create/delete keys in keychains
    a. Demonstrate attack by corrupted/modified packet
    b. Demonstrate replay attack

    c. Demonstrate attack where 3P-A tries to add/delete/modify key from AUTH key chain
    d. Demonstrate attack where 3P-A tries to add/delete/modify key from 3P-B
4. Use of keys in a keychain on behalf of a user
    a. Demonstrate attack where users attempts to perform an action not permitted by the policy
    b. Demonstrate attack where user exceeds limits (number of uses) on a particular action
    c. Demonstrate attack on use by primary user vs. other users
5. SP hardware protection
    a. Demonstrate attack on confidentiality and integrity of secure data in memory and registers during an interrupt and between calls to the TSM.

d) Structures

<u>Keychain structure</u> (see Figure 2)

- DRK + SRH
    - Derived Key from DRK
        - [KC_ID]
            - COUNTER
            - [KEY_ID]
                - KEY
                - KEY_POLICY
                    - POLICY_MATRIX
                    - PRIMARY_USER_ID

KC_ID (keychain IDs) are globally unique IDs, assigned by the authority. The authority can assign third parties one or more KC_IDs. The third parties can manipulate keys and policies within the key chains they are assigned. KEY_ID (key IDs) are unique within each keychain. Monotonically increasing counters are associated with each KC_ID and are maintained by the owner of the keychain is used by the TSM to prevent replay attacks on update messages.

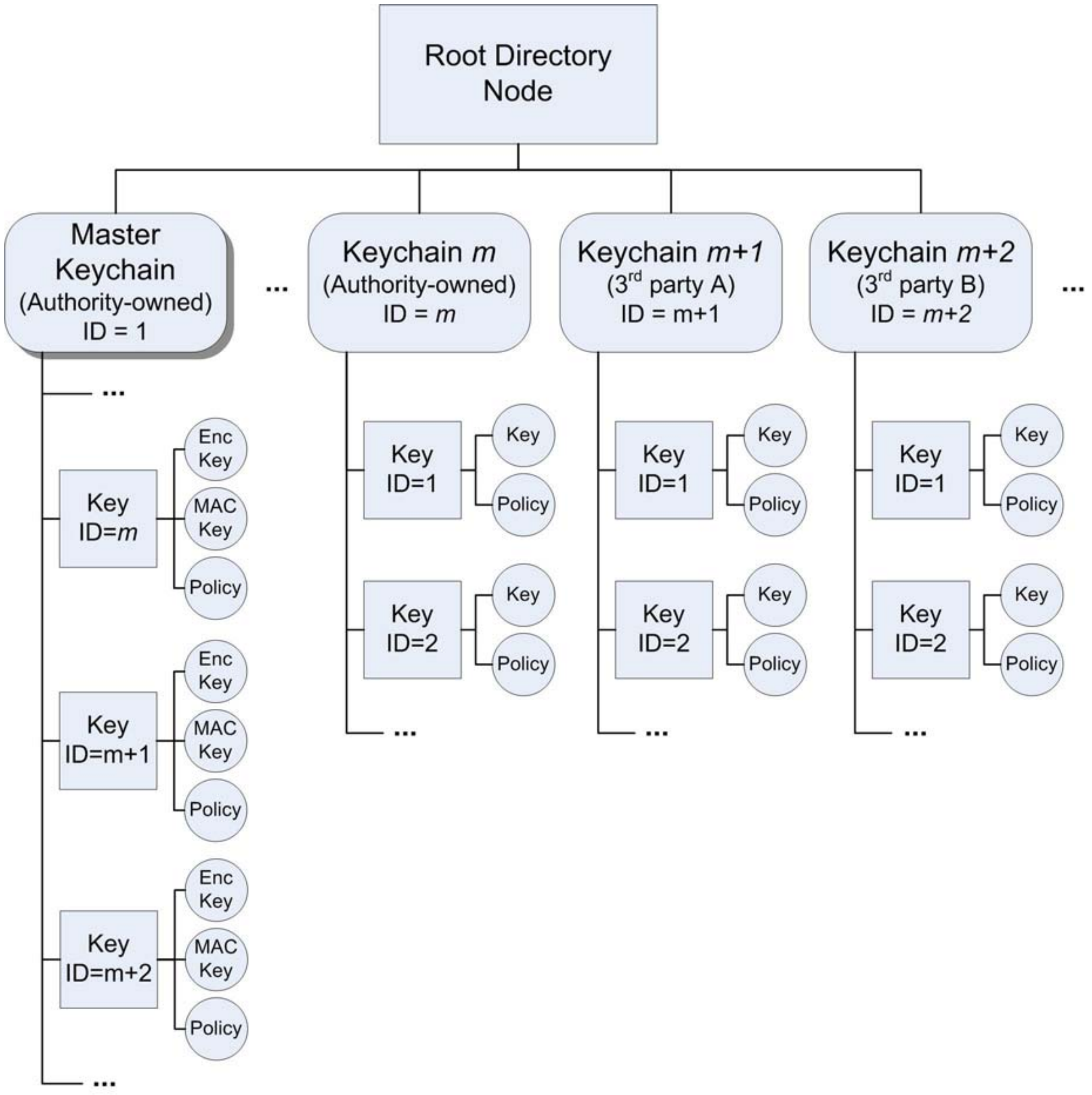
Root Directory Node
Master Keychain (Authority-owned) ID = 1
Keychain m (Authority-owned) ID = m
Keychain m+1 (3rd party A) ID = m+1
Keychain m+2 (3rd party B) ID = m+2
Key ID=m
Key ID=m+1
Key ID=m+2
Enc Key
MAC Key
Policy
Key ID=1
Key ID=2
Key
Policy

Figure 2: Keychain Structure

Figure 2 shows a sample keychain structure, where the authority owns and manages *m* keychains (including the master keychain), and the authority delegates access to other keychains (ID = *m*+1, *m*+2, …) to third parties. Each of the regular keychains (ID > 1) contains a list of keys, each with an associated policy (see Table 1). The TSM controls access to these keys, enforcing the attached policies. The TSM never permits the keys of the master keychain to be used by applications. They are only used to authenticate and decrypt update messages for the remaining keychains.

Policy Matrix

Table 1: Policy Matrix

Action	Primary User	Other Users	"Remaining Uses" validity	Remaining Uses
Encrypt	Allow/Deny	Allow/Deny	True/False	Integer
Decrypt	Allow/Deny	Allow/Deny	True/False	Integer
Re-encrypt	Allow/Deny	Allow/Deny	True/False	Integer
Generate MAC	Allow/Deny	Allow/Deny	True/False	Integer
Verify MAC	Allow/Deny	Allow/Deny	True/False	Integer
Gen. Session Key	Allow/Deny	Allow/Deny	True/False	Integer

Note: For the current demo, we only allow a policy for "Remaining Uses" and not a Start Date/Time and Expiration Date/Time. This simplifies the demo by not requiring the TSM to have a trusted time source. User authentication is provided by the OS through credentials passed-in when using keys.

Message structure – for *SPSLib_Process_ExternMsg().* See Table 2.

Confidentiality and integrity protected message from authority arrives and is interpreted by the TSM.

- P_KC_ID
  - Keychain ID
- P_MAC_NONCE *(KC_ID=1 only)*
  - For keychain add/delete commands on KC_ID=1, the nonce used to produce derived key for MAC of the message.
- P_ENC_NONCE *(KC_ID=1 only)*
  - For keychain add/delete commands on KC_ID=1, the nonce used to produce derived key for encryption of the E_* fields of the message.
- E_COMMAND
  - Encrypted.
  - Command: "Add Key" or "Delete Key". *(KC_ID≠1 only)*
  - Command: "Add Keychain" or "Delete Keychain" available only to authority. *(KC_ID=1 only)*
  - Modification of keys or policy is performed by deleting and re-adding a key.
- E_COUNTER
  - Encrypted.
  - Keychain Message Counter.
- E_KEY_ID *(delete only)*
  - Encrypted.
  - Key ID to delete.
- E_KEY_STRUCTURE *(add only)*
  - Encrypted.
  - Key structure to add, including one or more key nodes and a policy node (with primary user ID).
- P_MAC
  - MAC over the rest of the structure using a derived key for KC_ID=1 or the stored signing key otherwise.

e) Commands & Interfaces

The following are the commands that the third party or the authority can use to add and delete keys from their respective key chains, or for the authority to add or delete keychains.

Authority or third party:

- Add key to key chain
    - <"*TSM_Key_Add*", KC_ID, KEY_STRUCTURE>
- Delete key from key chain
    - <"*TSM_Key_Delete*", KC_ID, KEY_ID>

Authority only:

- Add keychain with specified authentication keys
    - <"*TSM_Keychain_Add*", KC_ID, KEY_STRUCTURE>
- Delete keychain
    - <"*TSM_Keychain_Delete*", KC_ID>

<u>Relevant TSM Interfaces</u>

SPSLib_Process_ExternMsg (void *memLoc, size_t size)

> This interface is a generic interface that can be used to add/delete keys and policies from the authority and third parties, as well as add/delete keychains from the authority. A signed message (see message structure above) is sent by the authorized party, and is verified by the TSM before the operation is completed.

Other interfaces, including "Actions" that make use of keys, are specified in the TSM API. See Table 2.

III. SP Library/TSM Interface API for SP Services

This TSM API is specifically targeted at the key-management application (rather than general purpose TSM interface for arbitrary applications). The SP Services Library should implement the following TSM_Interface API calls:

Table 2: SP Library/TSM Interface API

TSM API		Description
SPSLib_Process_ExternMsg (tsm_msg_t *message, size_t size, counter_t counter, msg_sig_t MAC)		Process a remote command from the authority or a third party to create/delete keys or keychains. The *message* structure specifies the command and its parameters as well as any nonces used to encrypt and sign the message. The counter prevents replay and the MAC verifies the integrity and source of the message. The authority uses DRK-derived keys to encrypt & sign messages to create/delete keychains or to enable/disable keychains. The authority and third parties use dedicated keys to create/delete keys on keychains they have authorization for.
TSM External Message Commands	TSM_Set_Emergency_Level (level_t emergency_level, unsigned int emergency_counter)	Authority sets the emergency level and emergency state of the device, to either EMERGENCY_NONE (0) for no emergency or a value in the range from EMERGENCY_LOW (1) to EMERGENCY_HIGH (*n*) for an active emergency. The new counter value is set and must be monotonically increasing from the previous value. The authority authenticates the message using nonces and DRK-derived keys.
	TSM_Keychain_Create (keychain_t keychain_ID, key_struct_t keys, level_t emergency_level)	Authority creates a new keychain, specifying the new ID and the key structure that will later authenticate access to the keychain. The emergency_level specifies the minimum emergency level that must be set on the device in order to use the keys on this keychain. The authority authenticates the message using nonces and DRK-derived keys.
	TSM_Keychain_Delete (keychain_t keychain_ID)	Authority deletes an existing keychain entirely, specifying its ID. The authority authenticates the message using nonces and DRK-derived keys.
	TSM_Keychain_Disable (keychain_t keychain_ID)	Authority disables the use of an entire keychain, regardless of any policies attached to the keys. The authority authenticates the message using nonces and DRK-derived keys.
	TSM_Keychain_Enable (keychain_t keychain_ID)	Authority enables the use of a keychain that has been previously disabled. The authority authenticates the message using nonces and DRK-derived keys.

	TSM_Key_Add (keychain_t keychain_ID, key_struct_t keys)	Add a new key structure onto an existing keychain (includes key + policy). The message is authenticated using the keychain-specific encryption and MAC keys.
	TSM_Key_Delete (keychain_t keychain_ID, key_t key_ID)	Delete an existing key from a keychain, specifying its ID. The message is authenticated using the keychain-specific encryption and MAC keys.
level_t SPSLib_Get_Emergency_Level (user_t user)		Retrieves the current value of the emergency level and state of the device. May be optionally limited to the TML only or require user authentication.
SPSLib_KeyOp_Encrypt (user_t user, keychain_t keychain_ID, key_t key_ID, void *data, size_t len)		Encrypt data with the specified key.
SPSLib_KeyOp_Decrypt (user_t user, keychain_t keychain_ID, key_t key_ID, void *data, size_t len)		Decrypt data with the specified key.
SPSLib_KeyOp_ReEncrypt (user_t user, keychain_t keychain_ID, key_t dec_key_ID, key_t enc_key_ID, void *data, size_t len)		Decrypt data with one key and encrypt with another key from the same keychain in one atomic operation. Requires permissions for “re-encrypt” on the decryption key and “encrypt” on the encryption key.
SPSLib_KeyOp_GenerateMAC (user_t user, keychain_t keychain_ID, key_t key_ID, void *mac, void *data, size_t len)		Generate a MAC over the data with the specified key.
int SPSLib_KeyOp_VerifyMAC (user_t user, keychain_t keychain_ID, key_t key_ID, void *mac, void *data, size_t len)		Verify an existing MAC over the data with the specified key. Returns a result for either a “MAC Match” or a “MAC Mismatch”
SPSLib_KeyOp_GenerateSessionKey(user_t user, keychain_t keychain_ID, key_t key_ID, void *sessionkey, void *nonce, size_t len)		Generate a new session key using the specified key and provided nonce data.
SPSLib_Load_SecureStorage (void *addr, size_t size)		Tells the TSM where the encrypted secure storage structure is located in memory.
size_t SPSLib_getSize_SecureStorage (void)		Returns the amount of memory necessary to save a copy of the current secure storage structure. Used to pre-allocate memory before calling SPSLib_Store_SecureStorage().
SPSLib_Store_SecureStorage (void *addr, size_t size)		Requests that the TSM write an updated copy of the secure storage structure to a memory location to be saved to disk. The TSM will also update the SRH register.

Note: For the demo, we only support symmetric keys and a limited set of key operations and policies.

IV. TML & SCOS Interfaces

To be defined by NPS and ISI.

The following SCOS interfaces are defined in [SCOSspec]:

- scos_flush_segment
  ```
  unsigned int scos_flush_segment( const selector_type selector );
  ```
- scos_read_next
  ```
  unsigned int scos_read_next(
          const unsigned int major, const unsigned int minor,
          const unsigned int num_requested,
          void * const buffer, unsigned int * num_read);
  ```
- scos_read_random
  ```
  unsigned int scos_read_random(
          const unsigned int major, const unsigned int minor,
          const unsigned int offset, const unsigned int num_requested,
          void * const buffer, unsigned int * num_read);
  ```
- scos_write_next
  ```
  unsigned int scos_write_next(
      const unsigned int major, const unsigned int minor,
      const unsigned int num_requested,
      const void * const buffer1, unsigned int * num_written);
  ```
- scos_write_random
  ```
  unsigned int scos_write_random(
      const unsigned int major, const unsigned int minor,
      const unsigned int offset, const unsigned int num_requested,
      const void * const buffer2, unsigned int * num_written);
  ```
- scos_shutdown
  ```
  void scos_shutdown( const char * const[3] message );
  ```
- scos_powerdown
  ```
  void scos_powerdown(void);
  ```
- scos_login
  ```
  unsigned int scos_login(const char * const username, const char * const password);
  ```
- scos_who
  ```
  void scos_who( char * const username );
  ```
- scos_logout
  ```
  void int scos_logout(void);
  ```
- scos_set_password
  ```
  unsigned int scos_set_password( const char * const password );
  ```

The following TML interfaces are defined in [TMLspec] (see also [TMLguide]):

- kio_printf
  ```
  void kio_printf(const char * const buffer);
  ```
- kio_printf_str
  ```
  void kio_printf_str(const char * const format, const char * const buffer);
  ```
- kio_printf_int
  ```
  void kio_printf_int(const char * const format, const int value);
  ```
- kio_printf_char
  ```
  void kio_printf_char(const char * const format, const char value);
  ```
- register state
  ```
  /* This structure defines the register state passed to the CEMInterrupt calls */
  typedef struct {
      /* The ‘plx’ fields below refer to the register state at the time the
      interrupt occurred. If the interrupt occurred inside PL0 the
      ‘ssplx’ and ‘espplx’ fields will contain 0 */
      unsigned int ssplx; /* the SS register at the time of the interrupt */
      unsigned int espplx; /* the ESP register at the time of the interrupt */
      unsigned int eflags; /* the flags register */
      unsigned int csplx; /* the CS register at the time of the interrupt */
      unsigned int eipplx; /* the IP register at the time of the interrupt */
      unsigned int error_code; /* the error code that caused the interrupt,
      only supported for interrupts
      0x08, 0x0A, - 0x0E, and 0x10,
      all other interrupts have 0 in this field */
      unsigned int int_num; /* the interrupt number */
      unsigned int eax; /* the EAX register */
      unsigned int ecx; /* the ECX register */
      unsigned int edx; /* the EDX register */
      unsigned int ebx; /* the EBX register */
      unsigned int esp; /* the ESP register */
      unsigned int ebp; /* the EBP register */
      unsigned int esi; /* the ESI register */
      unsigned int edi; /* the EDI register */
      unsigned int sspl0; /* the SS register in the interrupt handler */
      unsigned int cspl0; /* the CS register in the interrupt handler */
      unsigned int ds; /* the DS register */
      unsigned int es; /* the ES register */
      unsigned int fs; /* the FS register */
      unsigned int gs;
  }registers_struct;
  ```

V. SP Emulation Module API

The SecureCore demo will provide a software emulation of Authority-mode SP hardware.

SP Emulation Function summary:

Table 3: SP Emulation Module API

	Instr. Class	SP Instruction	SP Emulation API	Exported	Description
Authority-mode CEM	Initialize	drk.set.*sel*	SPHW_DeviceRootKey_Set	User Code	Set the DRK register.
		drk.lock	SPHW_DeviceRootKey_Lock	User Code	Set the DRK_Lock register, preventing DeviceRootKey_Set() until next PowerOn().
	Master Secrets / CEM Register Access	drk.derive	SPHW_DeviceRootKey_Derive	User Code	Generate a derived key using the DRK into the CEM Buffer.
		srh.get	SPHW_StorageRootHash_Get	User Code	Copy the SRH into the CEM Buffer.
		srh.set	SPHW_StorageRootHash_Set	User Code	Copy the CEM Buffer into the SRH.
		gr.set.*sel*	SPHW_GR_Set	User Code	Set a general register from one word of the CEM Buffer.
		gr.get.*sel*	SPHW_GR_Get	User Code	Set two words of the CEM Buffer from general registers.
	CEM	begin_cem.a	SPHW_BeginCEM_auth	User Code	Enter active Authority CEM. CIC checking is not emulated.
		end_cem	SPHW_EndCEM_auth	User Code	Exit the currently active Authority CEM task.
	Secure Memory	secure_load	*(see below)*	User Code	For emulation, explicit secure access to memory is not made on each load and store instruction. Instead secure memory areas are defined by address range and are protected when not in active CEM.
		secure_store	*(see below)*	User Code	
		N/A	SPHW_SecureArea_Add	User Code	Setup a new secure memory region. Secure region will be protected on interrupts and across CEM invocations.
		N/A	SPHW_SecureArea_Remove	User Code	Remove an existing secure memory region.

		N/A	SPHW_SecureArea_Store	User Code	Immediately protect an active secure memory region. Future accesses in CEM will be to ciphertext.
		N/A	SPHW_SecureArea_Load	User Code	Immediately restore an active protected region. Future accesses in CEM will be to plaintext.

Hardware Operations	Power	N/A	SPHW_PowerOn	TML Only	Initialize SP persistent state and emulation data structures.
		N/A	SPHW_PowerOff	TML Only	Save SP persistent state.
	Interrupt Handling	N/A	SPHW_CEMInterrupt_Suspend	TML Only	Emulate CEM operations for an interrupt.
		N/A	SPHW_CEMInterrupt_Resume	TML Only	Emulate CEM operations for resuming from interrupt.
	Faults	N/A	SPHW_CheckFault	TML Only**	Check the result of an emulated SP instruction to determine if a hardware fault was generated.

SP Emulation Function Header:

sp_emu.h:

```
/* general parameters */
#define INIT_SIZE 4096                    // size of data blob used to save SP hardware state

/* define register word size */
#define WORD_SIZE 4                       // word size of registers in bytes
typedef unsigned int gpreg_t;

/* define fault types – private – do not check directly */
typedef int SPFault;

/* secure memory areas */
typedef struct {
   void *addr;
   size_t size;
   unsigned int state;  /* 0 == plaintext, 1 == ciphertext */
   gpreg_t hash[4];
   gpreg_t iv[2]

} SPHW_SecureArea_t;


#define SPArea_DESTROY   1
#define SPArea_RELEASE   2
```

```
SPFault SPHW_DeviceRootKey_Set (unsigned int sel, const gpreg_t rs1, const gpreg_t rs2);
SPFault SPHW_DeviceRootKey_Lock(void);
SPFault SPHW_DeviceRootKey_Derive(void);
SPFault SPHW_StorageRootHash_Get(void);
SPFault SPHW_StorageRootHash_Set(void);
SPFault SPHW_GR_Set (unsigned int sel, gpreg_t *rd);
SPFault SPHW_GR_Get (unsigned int sel, const gpreg_t rs1, const gpreg_t rs2);
SPFault SPHW_BeginCEM_auth (void);
SPFault SPHW_EndCEM_auth (void);
SPFault SPHW_SecureArea_Add (void *addr, size_t size);
SPFault SPHW_SecureArea_Remove (const unsigned int opts, void *addr);
SPFault SPHW_SecureArea_Store (void *addr);
SPFault SPHW_SecureArea_Load (void *addr);

int SPHW_PowerOn (void *initdata);
int SPHW_PowerOff (void *initdata);
SPFault SPHW_CEMInterrupt_Suspend (void *regs, size_t regslen, void *returnip, const unsigned int
        compartmentid, const unsigned int processid);
SPFault SPHW_CEMInterrupt_Resume (void *regs, size_t regslen, void *returnip, const unsigned int
        compartmentid, const unsigned int processid);
int SPHW_CheckFault (SPFault fault);
```

SP Emulation Module requirements of the TML:

(updated with comments from Thuy, Paul, and Ganesha – To be updated with further discussion)

- Retrieve SP persistent data for PowerOn
    - On bootup, the TML will call *SPHW_PowerOn()*, followed by *SPHW_DeviceRootKey_Lock()*, as specified in Section VI.
    - For the initial implementation, the persistent data will be compiled-in statically, until the ability to read this data from a file is available.
- Store SP persistent data for PowerOff
    - Prefer data to be persistent, but could assign static data and not demonstrate policy/keychain changes across reboots. For the initial implementation, this persistent data will not be saved at PowerOff, until the ability to write files is available.
- Call CEMInterrupt_Suspend/Resume at the right times
    - Any time the TML gets an interrupt and will execute Supervisor/User code.
        - Initially, only test with one specific interrupt type for debugging/testing, then later enable for all interrupts.
    - Secure areas
        - Memory addressing. The SP emulation module will be able to read and write to user application memory.
    - Registers
        - Location and layout of registers is provided by TML. A pointer to the registers will be passed to these functions for modification by the emulation module.
            - Get an interrupt frame on stack. The TML saves them on interrupt entry point.
- Check SPFaults after each SP instruction is called
    - Pass to SCOS which handles fault in some way. Can crash the app or restart the app.
    - For the demonstration, SPFault return values will generally be exported to the user application, as is the CheckFault function. The application will check the result and handle the fault, allowing the demo application to test and respond to attacks. Only the TML-only functions have their SPFault return values checked and handled by the TML.
- Allocate secure TML memory for the SP Emulation Module to use
    - Declare statically in the Emulation module code with a constant limit of the number of areas at compile time.
- Debugging output
    - For the purposes of debugging, messages may be printed to an appropriate I/O device by the SP emulation module and by the TSM and demo application.
    - There should be some mechanism to capture the normal and debugging output of the program to a file, to allow for documentation and archiving of the SecureCore demonstration, as well as to assist further development and debugging efforts.
    - For the initial implementation, capturing of output to a file will not be provided, but instead a screen-capture can be used for the demo running inside a virtual machine.

SP Emulation Function Details:

SP Device Initialization: Exported

NAME

DeviceRootKey_Set
DeviceRootKey_Lock

SYNOPSIS

SPFault SPHW_DeviceRootKey_Set (unsigned int sel, const gpreg_t rs1, const gpreg_t rs2);
SPFault SPHW_DeviceRootKey_Lock(void);

DESCRIPTION

SPHW_DeviceRootKey_Set() is used for device initialization to configure a new DRK in a device. SPHW_DeviceRootKey_Lock() is used on bootup to prevent modification of the DRK on a device which has already been initialized.

SPHW_DeviceRootKey_Set() sets two words of the DRK register, by concatenating rs1|rs2. The sel parameter selects which portion of the DRK register to set: sel == 0, 2 for 32-bit word-size, selecting either the low 64-bits or high 64-bits respectively; sel == 0 for a 64-bit word-size, always setting the entire 128-bit register.

RETURN VALUE

SPFault (see SPHW_CheckFault() below)

CEM Operations: Exported

NAME

DeviceRootKey_Derive

SYNOPSIS

SPFault SPHW_DeviceRootKey_Derive (void);

DESCRIPTION

SPHW_DeviceRootKey_Derive generates a new derived key within the SP Emulation Module, using the DRK combined with the 128-bit nonce taken from the low-order bits of the CEM_Buffer register. The same derived key can be regenerated at a later time by specifying the same nonce, as long as the DRK remains unchanged.

RETURN VALUE

SPFault (see SPHW_CheckFault() below)

CEM Operations: Exported

NAME

StorageRootHash_Get
StorageRootHash_Set

SYNOPSIS

SPFault SPHW_StorageRootHash_Get (void);
SPFault SPHW_StorageRootHash_Set (void);

DESCRIPTION

SPHW_StorageRootHash_Get() and SPHW_StorageRootHash_Set() copy the entire SRH register to and from the CEM Buffer register, respectively. For SPHW_StorageRootHash_Set(), the data is copied atomically and the entire operation will either succeed or the SRH will remain unchanged.

RETURN VALUE

SPFault (see SPHW_CheckFault() below)

CEM Operations: Exported

NAME

GR_Get
GR_Set

SYNOPSIS

SPFault SPHW_GR_Get (unsigned int sel, const gpreg_t rs1, const gpreg_t rs2);
SPFault SPHW_GR_Set (unsigned int sel, gpreg_t *rd);

DESCRIPTION

SPHW_GR_Get() and SPHW_GR_Set() are used in CEM to access the CEM Buffer register.

SPHW_GR_Get() retrieves two words from general registers (rs1, rs2), concatenates them, and writes to a region of the CEM Buffer register. The sel parameter selects which portion (a pair of words) of the register to set: sel == 0, 2, 4, or 6 for 32-bit word-size, where 0 is the lowest 64-bits and 6 is the high 64-bits; sel == 0 or 2 for a 64-bit word-size.

SPHW_GR_Set() retrieves one word from the CEM Buffer register and sets a general register (rd). The sel parameter selects which word to get: sel == 0..7 for 32-bit word size; sel=0..3 for 64-bit word size.

Note: The CEM_Buffer is an intermediate register used for setting the DRK, reading and writing the SRH in one atomic operation, and creating and accessing derived keys.

RETURN VALUE

SPFault (see SPHW_CheckFault() below)

CEM Operations: Exported

NAME

BeginCEM_auth
EndCEM_auth

SYNOPSIS

SPFault SPHW_BeginCEM_auth (void);
SPFault SPHW_EndCEM_auth (void);

DESCRIPTION

SPHW_BeginCEM_auth() indicates the start of TSM code and switches the emulated processor into Active CEM mode (sets CEM_Mode = 01). Currently for the demo, no code integrity checking is done, but future prototypes may permit BeginCEM to take parameters indicating the location and size of the TSM code and hashes.

SPHW_EndCEM_auth() indicates the end of TSM code and switches the emulated processor out of CEM mode (sets CEM_Mode = 00).

RETURN VALUE

SPFault (see SPHW_CheckFault() below)

CEM Operations: Exported

NAME

SecureArea_Add
SecureArea_Remove
SecureArea_Store
SecureArea_Load

SYNOPSIS

```
SPFault SPHW_SecureArea_Add (void *addr, size_t size);
SPFault SPHW_SecureArea_Remove (const unsigned int opts, void *addr);

SPFault SPHW_SecureArea_Store (void *addr);
SPFault SPHW_SecureArea_Load (void *addr);
```

DESCRIPTION

SPHW_SecureArea_Add() and SPHW_SecureArea_Remove() manage secure storage memory regions for CEM execution. A secure memory region is defined by its starting address and size. All secure memory regions will be encrypted and hashed on an interrupt by the SP Emulation Module, before passing control to the OS or other applications. On resumption of CEM, the areas will be decrypted and integrity checked. Since secure areas are located in volatile memory, their contents is not persistent across reboots of the machine. Metadata is maintained inside the emulation module.

New areas are created using SPHW_SecureArea_Add(). Areas that are no longer used during the current TSM session can be removed using SPHW_SecureArea_Remove(). They can then either be destroyed (contents zeroed) or released in unencrypted form, as specified by the options.

opts for SPHW_SecureArea_Remove():

SPArea_DESTROY: zero the contents of the memory area
SPArea_RELEASE: keep the memory area decrypted and release contents

SPHW_SecureArea_Store() forces a secure memory region to be encrypted immediately. Future accesses, even in CEM, will access the ciphertext.

SPHW_SecureArea_Load() decrypts a previously stored region to resume plaintext access in CEM.

RETURN VALUE

SPFault (see SPHW_CheckFault() below)

Emulation Initialization: TML only

NAME

PowerOn
PowerOff

SYNOPSIS

int SPHW_PowerOn (void *initdata);
int SPHW_PowerOff (void *initdata);

DESCRIPTION

Emulation-specific initialization and shutdown routines. SPHW_PowerOn() initializes internal data structures and simulates hardware reset of device state. It reloads all non-volatile device state from initdata. SPHW_PowerOff() cleans up any internal data structures and saves the non-volatile state to initdata.

initdata should be NULL at SPHW_PowerOn() for a new/clean device with no state (i.e. no saved DRK or SRH – clean from the factory). Otherwise, initdata should be saved on SPHW_PowerOff() and provided unchanged on the next SPHW_PowerOn(). All non-volatile data will be saved between power cycles in this data blob of size INIT_SIZE.

RETURN VALUE

returns -1 on error or 0 for success.

SP Hardware Emulation: TML Only

NAME

CEMInterrupt_Suspend
CEMInterrupt_Resume

SYNOPSIS

SPFault SPHW_CEMInterrupt_Suspend (void *regs, size_t regslen, void *returnip, const unsigned int compartmentid, const unsigned int processid);
SPFault SPHW_CEMInterrupt_Resume (void *regs, size_t regslen, void *returnip, const unsigned int compartmentid, const unsigned int processid);

DESCRIPTION

SPHW_CEMInterrupt_Suspend() is called by the TML after any interrupt/exception/fault occurs, before turning control over to OS. It assumes registers have already been moved to a location in memory specified by regs with length regslen bytes. The return instruction pointer for the interrupt is specified in returnip, as well as the current compartment ID and process ID if known or NULL otherwise.

SPHW_CEMInterrupt_Suspend() causes CEM mode to be suspended if active, entering the Interrupted CEM state, and triggers the encryption of all general purpose registers and secure memory regions. If CEM mode was not active, it has no effect.

SPHW_CEMInterrupt_Resume() is called by the TML immediately before returning from any interrupt/exception/fault. If it detects a return to the previously suspended CEM thread, it will decrypt the register contents and secure memory regions, performing an integrity check. Then Active CEM mode is resumed. If not returning to the suspended CEM thread, it has no effect.

RETURN VALUE

SPFault (see SPHW_CheckFault() below)*

** Note: Since these functions are not exported to user code, even for the demonstration, the TML must check the return value with SPHW_CheckFault() and in turn crash the application or halt if a fault occurs.*

<u>SPFault Functions:</u> TML Only**

NAME

CheckFault

SYNOPSIS

int SPHW_CheckFault (SPFault fault);

DESCRIPTION

SPHW_CheckFault() checks the result of an SPFault variable returned by an SP Emulation function to determine if the emulated instruction generated a hardware fault.

RETURN VALUE

Returns 0 on success (no fault occurred) or a positive value indicating the fault type (refer to SP Architecture Manual [SPArch] – SP Exceptions/Faults Section).

*** Note: By design, SPHW_CheckFault() need not be exported to user code. Instead the TML should check the result of each emulated SP instruction and trigger an actual fault/exception to the OS & user application, or crash the user application, as needed. However, due to practical considerations for the demonstration, this function and the return value of each instruction emulation function will be exported to user code, and the return value will be checked by the calling application. The application will in turn handle the fault or halt when an error occurs. This allows the demo application to attempt attacks, detect the resulting fault, and continue with the rest of the demo.*

VI. Demonstration Details

This section provides additional technical details of the demonstration procedure, including a step-by-step initialization procedure and more information about the implementation of each step of the demonstration. It also provides a list of data elements used in the demonstration. For the sequence of key-management operations in the demonstration, refer to the overview in Section II.

Important data elements of key management application:

- Keychains & Keys (Initial Secure Storage plus those added during operation)
    - KC01 (Authority Master Keychain)
        - K02 (3P-A access keys)
        - K03 (3P-A access keys)
        - K04 (3P-B access keys)
    - KC02 (3P-A existing)
        - K01 “3P-A-KeyA” (User_*, Enc)
        - K02 “3P-A-KeyB” (User_*, Dec)
        - K03 “3P-A-KeyC” (User_*, Enc/Re-encrypt)
    - KC03 (3P-A new)
        - K01 “3P-A-Key1-User_X” (User_X, Enc/Dec)
        - K02 “3P-A-Key2-User_Y” (User_Y, Enc/Dec)
    - KC04 (3P-B new)
        - K01 “3P-B-Key1-User_*” (User_*, Enc)
        - K02 “3P-B-Key2-User_*” (User_*, Dec)
- Auth/3P* key-management commands (Source/signature, Keychain, Keys, Command)
    - (Auth, KC03, 3P-A access keys, ‘Keychain_Create’)
        - Also a corrupted version
    - (Auth, KC04, 3P-B access keys, ‘Keychain_Create’)
    - (3P-A, KC05, 3P-A access keys, ‘Keychain_Create’)
    - (3P-A, KC03, 3P-A-Key1-User_X, ‘Key_Add’)
    - (3P-A, KC03, 3P-A-Key2-User_Y, ‘Key_Add’)
    - (3P-A, KC02, K02, ‘Key_Delete’)
    - (3P-B, KC04, 3P-B-Key1-User_*, ‘Key_Add’)
        - Also a corrupted version
    - (3P-B, KC04, 3P-B-Key2-User_*, ‘Key_Add’)
    - (3P-A, KC01, Fake-Key1, ‘Key_Add’)
    - (3P-B, KC03, K01 “3P-A-Key1”, ‘Key_Delete’)
    - (3P-A, KC03, encrypted copy of 3P-B-Key1. ‘Key_Add’)
- Secure storage
    - SecureStorage-current
    - SecureStorage-original
- Secure areas
    - SecureArea

Additional implementation and technical details for the demonstration of the TSM and SP emulation functions:

- Step 1: Initialization
    - Boot and initialization sequence
        - As a part of the boot-up initialization routine, the TML must initialize SP HW emulation using the *SPHW_PowerOn()* function. After the successful execution of the *SPHW_PowerOn()* instruction, the SP emulated HW must be in a state from which CEM state can be entered.
        - After boot-up/PowerOn, the TML should call *SPHW_DeviceRootKey_Lock()* to prevent the SCOS or user applications from resetting the DRK.
            - This call can be made from either user-mode or kernel-mode code, and for the demo may instead be performed by the SCOS during its initialization, or from the key-management application on startup. However in a full implementation this should be done by trusted software or a secure BIOS during bootup, before executing any untrusted code.
    - Initialization of key-management application and TSM (including)
        - The demo application, keyManagementApp.bin, is invoked manually. The application contains a TSM which is used for all access to and use of protected keys. Each time the TSM is invoked by the application through one of the *SPSLib_** interfaces, *SPHW_BeginCEM_auth()* is called to enter CEM mode, followed by *SPHW_EndCEM_auth()* to exit CEM mode after the TSM operation is complete.
        - As a part of the initialization of the TSM, *SPHW_SecureArea_Add*() is called to register the memory regions that the TSM will use for storing sensitive intermediate data.
            - The TSM will use statically allocated memory and will not call any TML/SCOS interfaces to allocate memory. This function call will register the memory location and size with the SP emulation module so that encryption and decryption of the SecureArea can be performed on interrupts.
                - The memory location and size is with respect to user space, and the kernel code (SP emulation module) must be able to touch and operate on this space when an interrupt occurs or CEM mode is changed.
        - Generally the following operations on SecureAreas are performed when CEM is invoked, interrupted, resumed, or ended
            - *SPHW_BeginCEM_auth()*: If any SecureAreas are registered with the SP emulation module, then this instruction will atomically check the integrity of the SecureAreas and decrypt them, while entering CEM mode.
            - *SPHW_CEMInterrupt_Pause()*: In addition to handling the other CEM interrupt related data, this instruction should encrypt the SecureAreas and store their hashes in the data structure in the emulation module.
            - *SPHW_CEMInterrupt_Resume()*: In addition to handling the other CEM interrupt related data, this instruction should atomically check the integrity of the SecureAreas and decrypt them.
            - *SPHW_EndCEM_auth()*: This instruction should encrypt any SecureAreas and store their hashes in the data structure in the emulation module before exiting CEM mode.

    - Load Secure Storage (SecureStorage-current), containing keys and their policies, into the TSM by the application using the function *SPSLib_Load_SecureStorage()*. The application pre-loads the Secure Storage into memory and then specifies the location and size to the TSM.
        - If a filesystem is supported, the SecureStorage file on disk is used as a parameter while invoking the keyManagementApp.bin executable. If file systems are not supported, the SecureStorage is compiled in statically with the key-management application.
        - The TSM moves the entire Secure Storage into a SecureArea after it is loaded. It then generates a derived key using *SPHW_DeviceRootKey_Derive()* and uses it to decrypt the structure. It also checks the integrity against the SRH, which is fetched using *SPHW_StorageRootHash_Get()*. The storage structure remains protected in the SecureArea across future calls to the TSM functions, so this initial processing only needs to be done once during initialization.
            - Accessing the generated derived keys and setting/getting the SRH also requires the *SPHW_GR_Set()* and *SPHW_GR_Get()* functions to access the intermediate hardware buffer register.
        - The TSM will continue to read and modify the copy of Secure Storage stored in the SecureArea in memory, rather than the copy on disk, as TSM operations are performed. For the demo, the copy on disk will only be updated when the key-management application requests that it be stored to disk, at which time the SRH register will also be updated.
- Step 2: Manage keychains
    - The authority is the only entity that can create/delete keychains. This is enforced by the TSM. The authority sends cryptographically verifiable messages to the device to manage keychains. For the demo, these messages are pre-computed, signed, and stored in the key-management application.
    - The key-management applications uses the *SPSLib_Process_ExternMsg()* interface to push the received/saved messages into the TSM. If a message is found to be authentic, the command embedded in the message is executed, as defined in Table 2.
    - Selection of keys to decrypt data and verify hashes in the message is done using internal interfaces of the TSM. Messages from the authority are verified using derived keys from the DRK.
    - Management of keychains includes commands to create/delete keychains for the authority and third parties, resulting in modifications to the Master Keychain.
- Step 3: Manage keys in a keychain
    - The authority or third party can send messages to add or delete keys their respective keychains. These messages are encrypted and signed using the keys on the Master Keychain corresponding to each regular keychain, which are shared with the third party owner of that keychain.
        - These messages are also pre-computed for the demo, and are passed into the TSM using the *SPSLib_Process_ExternMsg()* interface.
    - Since all messages are cryptographically verified, any attempt to modify keys in keychains not owned by the sender of the message is detected by the TSM. For the demo, we use corrupted messages to demonstrate resilience against attacks in which 3P-A or 3P-B attempt to modify the keys in the key chains that belongs to each other or the authority.
    - Since the key material in the messages is encrypted using keys specific to each keychain, encrypted key material intended for one keychain cannot be copied into a message for another keychain and be correctly decrypted and accepted by the TSM.

- Step 4: Use of keys in a keychain on behalf of a user
    - The key-management application uses the *scos_login()* interface to authenticate users of the device. It then passes credentials via an authorization token to the TSM when making calls to use keys.
        - These credentials are supplied by the SCOS. For the demo, they will consist of the output from *scos_who()*. In a full-scale implementation, they would also include a signature from the SCOS using a key it shares with the TSM. Alternatively, the TSM could directly and securely query the SCOS to determine the currently authenticated user.
    - The key-management application requests the use of keys in a keychain on behalf of a user.
        - The key-management application calls the *SPSLib_KeyOp_** interfaces to make use of keys. It specifies a particular keychain and key. The stored policy determines if access is permitted.
        - If the policy for the requested action is allowed and specifies a valid "Remaining Uses" policy, then the action will be denied if Remaining Uses <= 0. If valid and allowed, the Remaining Uses is decremented upon completion of the action.
        - The caller specifies the user (via authorization token) for enforcement of the OS's access control policies by the TSM. If OS cannot be trusted or the credentials cannot be verified, all accesses are made as if by "Other users" and the remaining policies are enforced by the TSM.
- Step 5: Store updated Secure Storage to disk
    - Save modified keychains in Secure Storage and update SRH
        - Determine the space required to save the modified Secure Storage using *SPSLib_getSize_SecureStorage().* Verify that this does not exceed the amount of storage allocated or available on disk and in memory for storing the encrypted structure.
        - Request that the TSM write the updated encrypted contents of Secure Storage to unprotected application memory, using the function *SPSLib_Store_SecureStorage().*
            - The TSM will generate new derived keys to encrypt and MAC the contents of the Secure Storage structure, and may also reuse some existing derived keys for this purpose for parts of the structure that did not change.
            - During this process, the TSM will also update the SRH register using *SPHW_StorageRootHash_Set()*.
            - The encrypted blob is copied out of SecureArea to a memory location specified in the arguments. The SecureArea is then deleted, and will no longer be protected by the SP emulation module.
        - The application flushes the new Secure Storage blob to disk (SecureStorage-current), if flushing is implemented in the TML and SCOS.
    - Reload the original version of Secure Storage (SecureStorage-original) and demonstrate a Secure Storage replay attack
        - The application maintains two copies of Secure Storage (-current and -original).
            - Initially, before the demo, the two copies are identical. SecureStorage-current is then used for the demo, modified, and saved back to disk/memory. SecureStorage-original is then used to demonstrate this attack.
        - After the above demo, load SecureStorage-original using *SPSLib_Load_SecureStorage().*
            - The hash check should fail when comparing the old Secure Storage structure to the updated SRH, resulting in the TSM reporting an error.

- Step 6: Secure memory attacks
    - Real SP hardware allows the TSM to tag memory as ‘secure’ when accessing them in CEM mode, on a per-cache-line basis. It then automatically encrypts and MACs these tagged cache-lines as they are later evicted to main memory. This feature is emulated in the demo in software using SecureAreas. A SecureArea is a region of memory dedicated to the TSM, available in plaintext during CEM mode and then encrypted and hashed by the emulation module when CEM ends or is interrupted. The hash is stored internally in the emulation module, safe from attack by non-kernel software, and is checked when CEM is resumed. The SecureArea is then decrypted if the hash is successfully verified.
    - The key-management application attempts attacks on encrypted SecureArea data in-between calls to the TSM
        - The application attempts to read data stored in the SecureArea to show that only ciphertext is available. The attack can be re-run with the SecureArea protection manually removed by the TSM to show that sensitive data would otherwise be revealed.
        - The application modifies ciphertext, which is detected by the emulation module on the next TSM operation.
        - The application saves a region of valid SecureArea ciphertext, calls a TSM operation which changes that region, replays the old ciphertext, and calls another TSM operation, which causes the emulation module to detect the change.
    - The above attacks are also demonstrated during an interrupt of CEM
        - A software interrupt is triggered by the TSM with a special interrupt type. This in turn triggers *SPHW_CEMInterrupt_Pause()*, which upon detecting the interrupt type, performs the attacks, which are detected on return to CEM in the *SPHW_CEMInterrupt_Resume()* function.
        - In addition, attacks are performed during interrupt on the saved and protected general registers from CEM.
- Step 7: Attacks on SP hardware registers
    - The key-management application, outside of the TSM, will attempt to set the DRK, and to access CEM-only instructions (emulated functions). These will each trigger an appropriate SPFault.

VII. References

[SCDemo] "SecureCore Proof of Concept Demonstration". SecureCore working note. Sept. 9, 2007.

[SPArch] Jeffrey Dwoskin, Ruby Lee. "SP Processor Architecture Reference Manual". Princeton University Department of Electrical Engineering Technical Report CE-L2007-008. Draft version 0.7, November 21, 2007.

[SCOSspec] Paul C. Clark, Cynthia E. Irvine, Thuy D. Nguyen, Timothy E. Levin, Timothy M. Vidas, David Shifflett. "SecureCore Software Architecture: SecureCore Operating System (SCOS) Functional Specification". SecureCore Technical Report NPS-CS-08-006, November 30, 2007.

[TMLspec] David J. Shifflett, Paul C. Clark, Cynthia E. Irvine, Thuy D. Nguyen, Timothy M. Vidas, Timothy E. Levin. "SecureCore Software Architecture: Trusted Management Layer (TML) Kernel Extension Module Interface Specification". SecureCore Technical Report NPS-CS-07-021, December 20, 2007.

[TMLguide] David J. Shifflett, Paul C. Clark, Cynthia E. Irvine, Thuy D. Nguyen, Timothy M. Vidas, Timothy E. Levin. "SecureCore Software Architecture: Trusted Management Layer (TML) Kernel Extension Module Integration Guide". SecureCore Technical Report NPS-CS-07-022, December 20, 2007.