2007-12

SecureCore Software Architecture: SecureCore Operating System (SCOS) functional specification

Clark, Paul C.

Monterey, California, Naval Postgraduate School

NPS-CS-07-018

Trustworthy Commodity Computation
and Communication

SecureCore Technical Report

SecureCore Software Architecture:
SecureCore Operating System (SCOS) Functional Specification

Paul C. Clark, Cynthia E. Irvine, Thuy D. Nguyen,
Timothy E. Levin, Timothy M. Vidas, David J. Shifflett

December 2007

The Center for Information Systems Security Studies and Research

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0430566 and CNS-0430598 with support from DARPA ATO. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or of DARPA ATO.

Author Affiliation:

Center for Information Systems Security Studies and Research
Computer Science Department
Naval Postgraduate School
Monterey, California 93943

Table of Contents

[THIS PAGE IS INTENTIONALLY BLANK]

1 Introduction

1.1 Background

SecureCore is a research project funded by the National Science Foundation (NSF) to investigate the fundamental architectural features required for trustworthy operation of mobile computing devices such as smart cards, embedded controllers and hand-held computers. The goal is to provide secure processing and communication features for resource-constrained platforms, without compromise of performance, size, cost or energy consumption. In this environment, the security must also be built-in, transparent and flexible.

This document describes the interface to the SecureCore operating system (SCOS) for Phase 0 of the SecureCore project. Phase 0 is a rapid prototype with only enough specified functionality to demonstrate progress and potential capabilities of the SecureCore project. The SCOS interface includes a software emulation of the Secret-Protecting (SP) processor extensions [1]. Because this is a Phase 0 functional specification, this document is considered to be a rapid-prototype specification within a spiral life cycle model. This document is expected to change as experience is gained with the Least Privilege Separation Kernel (LPSK) environment or as better approaches are identified.

A description of the software architecture and definitions can be found elsewhere [2]. This document assumes the reader is familiar with the architecture and terminology of the SecureCore project.

1.2 Limitations

The Phase 0 implementation will incorporate only a subset of the envisioned SecureCore functionality, and will not be developed with high assurance techniques or covert channel analysis. The following lists the limitations on the SecureCore functionality implemented in Phase 0:

- The LPSK runs in PL0 without using any hardware support to create virtual machines.
- The SCSS runs in PL0.
- The SCOS executes in privilege level 1 (PL1), and applications execute in PL3.
- There is only one process per partition.
- There is only one application in PL3 per partition.
- A limited number of active partitions are supported.
- The trusted path application (TPA) is not supported.
- Detection of the invocation of the secure attention key (SAK) will be handled in PL0 and will not be reflected outside of PL0.
- There is no support for the setting of a session level.
- The keyboard and screen are the only exported devices. They will be attached by the LPSK during initialization.
- Graphics mode is not supported for the screen.
- Only one hard disk is supported.

- Networking is not supported.
- Non-blocked I/O is the only mode supported for exported devices (i.e., blocked I/O is not supported). For example, if an SCOS call is made to read from the keyboard device, and there is no data in the keyboard buffer, then an error is returned to the caller. Applications will therefore need to poll devices to determine if data is available.
- Creation of persistent memory objects during run-time is not supported.
- A file abstraction for accessing memory objects is not supported.
- Handheld devices are not supported.
- All the device driver support resides in PL0.
- Sensitivity labels are not supported.

1.3 Phase 0 Process Abstraction

The process abstraction is created by the LPSK. There is only one process per partition. Therefore, a process consists of all the subjects executing within a partition. There can be up to three subjects per process. The LPSK schedules processes in a round-robin fashion with a configurable fixed time slice.

All applications execute in PL3, with one application per partition. The Phase 0 SCOS does not support the scheduling of multiple applications within a partition.

1.4 Approach to Authentication in Phase 0

The SCOS provides a service that allows an application to authenticate a user by means of a password. It is the responsibility of the application to prompt the user for a username and password, then request authentication from the SCOS. A SecureCore computer is intended to support one physical user at a time, so the SCOS will only support one logged in user per process. Because there is no communication between SCOS instantiations, there is no concept of single-sign-on for the SecureCore computer in Phase 0. This may require a user to log in more than once if authentication is required by more than one application.

A separate password database should be maintained by each instantiation of the SCOS, otherwise there is the potential for password inconsistency and covert channels. The authentication data is stored in a password file as a hash of the expected password for each user.

When a user is successfully authenticated, the SCOS returns a success code to the calling application. The consequence of a successful authentication or logout is application-dependent.

1.5 Secondary Storage in Phase 0

The use of secondary storage is limited in Phase 0. All secondary storage objects that are needed by any subject must exist prior to LPSK initialization. The creation of secondary storage objects during run-time is not supported. However, there may be support in Phase 0 for flushing (i.e., updating) a memory segment to its source object on secondary storage. The flush system call is a low priority implementation item and may not be implemented in Phase 0.

The boot loader in Phase 0 is based on an open-source utility called *grub* [3]. It is used to load the LPSK into memory, and to perform additional LPSK initialization functions. A particularly useful feature of grub (for Phase 0) is the ability to read many different file system formats, which will allow Phase 0 executables and data to be placed in a familiar environment. For example, the PL3 applications can be stored in a FAT32 file system with a location such as "/applications/shell.out".

1.6 Referencing Devices

Device references with the SCOS will take the form of a (major, minor) tuple, where "major" refers to a category of devices, and the minor number refers to a specific instantiation of a device. The keyboard and monitor devices are logically attached by the LPSK during initialization of all active partitions, with the associated (major, minor) tuples passed to each PL during their initialization. Phase 0 will not support the dynamic attachment of other devices.

1.7 Partition Focus

The Trusted Management Layer (TML) detects the invocation of the secure attention key (SAK). The use of the SAK in Phase 0 is to signal the TML to change the partition with focus. Partition focus is changed on a round robin scheduling algorithm. For example, if partition0 has focus, then invoking the SAK will give focus to partition1. Invoking the SAK again will give focus to partition2, etc. The SAK in Phase 0 is "Alt-+" (i.e., the 'Alt' key and the '+' key pressed at the same time).

1.8 Executable Requirements

In order for the LPSK to load an executable file from secondary storage and prepare it for execution the following requirements must be met:

- The executable file must be constructed in the 32-bit Linear Executable (LX) file format. The development tools will generate this format when they are installed and configured per the instructions in a separate document.
- The first executable instruction must be at offset 0. This is dependent on the developer to write and link the code in a given order. Detailed instructions will be provided in a separate document.

1.9 Segments

The Phase 0 abstraction for memory objects is a segment. Segments are memory objects that are created by the LPSK during initialization and are accessible by the applications that are intended to use them, as directed with a configuration vector that is described in the next section.

There are four kinds of segments supported by the LPSK:

- Default Segments that are created by the LPSK as a result of loading an executable file are referred to as default segments. Default segments

	are the code segments, data segments and stack segments that are declared in the executable file.
• Dseg	A dseg is an additional data segment that is initialized from a non-executable file. In other words, it is data that is not compiled into the executable file. For example, an application may need to access data that may change over time, such as a list of valid e-mail addresses. Therefore, it may be wise to put the data into a separate file rather than compile it into the executable. Selectors to the optional dsegs are provided by the SCOS to the application, as described in Section 3.
• Mseg	An mseg is an un-initialized data segment. A typical use of an mseg is the allocation of run-time memory that cannot be compiled into an executable file. Selectors to the optional msegs are provided by the SCOS to the application, as described in Section 3. Msegs are not persistent across system reboots.
• Gseg	A gseg is a gate segment. It is used to provide call gates between privilege levels. PL3 application developers do not need to specify gsegs.

1.10 Interface Categories

The SCOS interfaces are separated into four parts: the PL3 initialization database, the PL3 configuration database, native call interfaces, and SP emulation call interfaces.

The PL3 initialization database refers to a file that is used to configure the initial run-time environment for applications.

The PL3 configuration database refers to information that is provided by the SCOS to an application during the initialization of the application. An application uses the PL3 configuration database to support its own functionality (e.g., the major and minor numbers for the attached devices).

A native call interface is an SCOS run-time request that is accessible by applications.

SP emulation call interfaces are run-time requests meant to support an emulation of some of the SP hardware features for demonstration purposes. The emulation calls are accessible by applications.

2 PL3 Initialization Database

Each privilege level has its own initialization database. These databases are used to create a configuration vector for the LPSK. This is shown graphically in Figure 1.

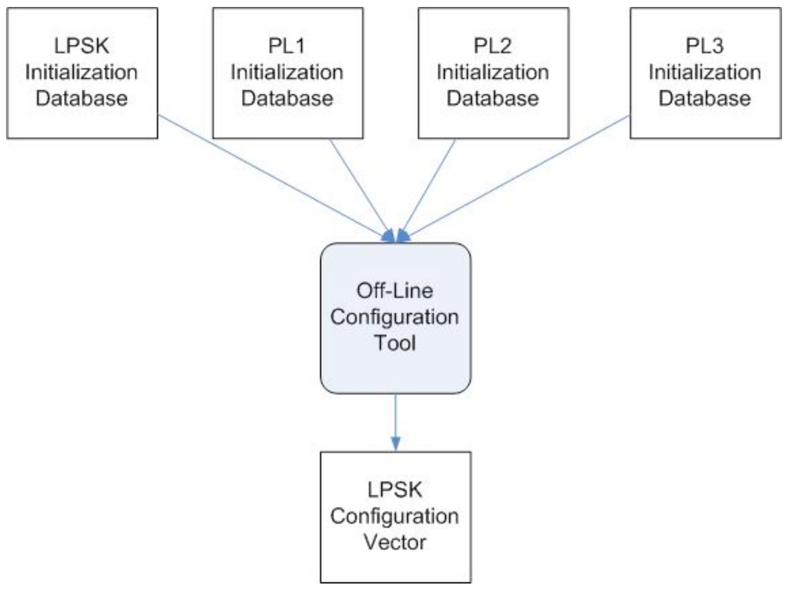
LPSK Initialization Database
PL1 Initialization Database
PL2 Initialization Database
PL3 Initialization Database
Off-Line Configuration Tool
LPSK Configuration Vector

Figure 1. Generation of the LPSK Configuration Vector

The PL3 portion of the configuration vector is used by the LPSK to initialize PL3 in a way that will start the applications appropriately. An off-line tool parses the input initialization databases and creates a binary configuration vector that is used by the LPSK during initialization. Using text files for the initialization databases allows for easy configuration changes and change tracking.

Comments within the PL3 initialization database may be inserted by preceding a string with a ‘#’. All text that appears after a ‘#’ and up to the end of the line is considered a comment and is not processed by the off-line tool. The end of the line is given by the ASCII line-feed character (0x0a). Blank lines are also considered comments, i.e., lines that consist only of ASCII tabs (0x09) and space (0x20) ending in the line-feed character).

The following syntax is used first to declare the location of application executables on secondary disk, which allows the LPSK to find them and load them into memory.

CODE[*partition*] = *path*;

“CODE” is a reserved word. The other parameters in the above syntax are explained below:

- *partition* is a number representing the partition the associated code should be loaded in (e.g., 0, 1, 2, etc.), as defined in the LPSK initialization database.
- *path* is a string that provides the executable’s location on secondary storage. If the location has white space, then it must be quoted. The boot loader in Phase 0 can read most file systems (e.g., FAT32), so the path can contain typical naming conventions and directory structures. There can be only one PL3 executable per partition in Phase 0. When loaded into memory, the LPSK enforces a policy that does not allow other partitions to access the code segment. The same policy is

enforced for the associated stack and data segments. In addition, all associated data and stack segments are not marked as executable.

The next part of the configuration declares any pre-existing application data that needs to be loaded into desgs from secondary storage. The LPSK allows dsegs to have either read only (RO), read/write (RW) or no access (NA) permissions associated with them. In addition, it is possible to set RO and RW permissions for applications running in partitions other than where the data resides (as long as the permissions do not conflict with information flow rules enforced by the LPSK between partitions). The syntax is as follows:

DSEG[*index*] = { *path, partition, permission*+ };

“DSEG” is a reserved word. The other parameters in the above syntax are explained below:

- *index* is an increasing array value starting from ‘0’.
- *path* is the directory/name of the file containing the required data. If the path contains white space it must be quoted.
- *partition* is the location that the PL3 dseg will be created, and into which the data from the above file will be copied. The partition is the “hosting” partition for the dseg. Valid partition numbers are defined in the LPSK initialization database.
- *permission*+ is a comma-separated list of per-partition permissions, where the first permission is associated with the 0th partition, the second permission is associated with the 1st partition, etc. There must be an assigned permission for each partition (as declared in the LPSK initialization database). The active partition to which the dseg is assigned (allocated) cannot have a NA permission to that dseg. If a non-NA permission is given to a partition, then the address space of the associated process will be updated to reflect the given permission.

The next part of the configuration declares msegs that are needed by the application. The syntax is as follows:

MSEG[*index*] = { *size, partition, permission*+ };

“MSEG” is a reserved word. The other parameters in the above syntax are explained below:

- *index* is an increasing array value starting from ‘0’.
- *size* is the size of the segment needed. The number can be given as the total number of bytes or with “KB”, “MB” and “GB” shortcuts, where a kilobyte is 1024 bytes, a megabyte is 1024KB and a gigabyte is 1024MB.
- *partition* is the location that the PL3 segment will be created. The partition is hosting the segment. Valid partition numbers are defined in the LPSK initialization database.
- *permission*+ is a comma-separated list of per-partition permissions, where the first permission is associated with the 0th partition, the second permission is associated with the 1st partition, etc. There must be an assigned permission for

each partition (as declared in the LPSK initialization database). There must be at least one active partition with RW access. If a non-NA permission is given to a partition, then the address space of the associated process will be updated to reflect the given permission.

The following is an example of what a PL3 initialization database file might look like. The following assumes there are two active partitions (0 and 1) and one passive partition (2), as would be defined in the LPSK configuration database.

```
# filename: pl3_config.txt
#
# This file is used to configure PL3 for all partitions.
#
# Created: 2007-11-07 (P. Clark)

# The following declares the executables.  The index refers to the assigned
# partition.
CODE[0] = /shell.out;
CODE[1] = /emergency.out;

# Other data segments required by the above applications:
#                                          Partitions
#                                          0    1    2
DSEG[0] =  { "/passwords.txt",      0, RW, NA, NA };
DSEG[1] =  { "/clearance.txt",      0, RO,  NA,  NA };
DSEG[2] =  { "/contacts.txt",       1, NA, RW, NA };
DSEG[3] =  { "/stuff.txt",          2, RO, RO,  NA };

# Additional run-time memory required by applications
#                              Partitions
#                              0    1    2
MSEG[0] = { 1MB, 0,       RW, NA, NA };
```

Note that the DSEG and MSEG declarations are optional; they are only necessary if an application requires data and memory that is not compiled with the application executable. Note that there is a maximum number of msegs, and a maximum number of dsegs that can be declared for a SecureCore computer. These are declared in the constants MAX_MSEGS and MAX_DESGS, respectively.

3 PL3 Configuration Database

This section describes the location and format for the PL3 configuration database provided by an instantiation of the SCOS to an application.

A selector to the PL3 configuration database will be passed to an application on its stack. After the SCOS has finished initializing itself and the environment, it will perform an initial outer-ring return to the application code. The application will construct a pointer from the passed selector to reference its configuration database.

The following C-code defines the constants and structures of the PL3 configuration database. A variable of type “int” is assumed to be 32 bits, and a variable of type “short int” is assumed to be 16 bits. Some of the types and constants used below assume the inclusion of lower-layer header files, e.g., the equivalent of an “lpsk.h”, to be defined at a later date.

```
#define MAX_USERNAME           32
#define MAX_PASSWORD           32

// Call interface return values
#define SCOS_AUTH_FAILURE      300
#define SCOS_LOGIN_BUSY        301
#define SCOS_INVALID_STRING    302
#define SCOS_NO_LOGIN          303

// The following structure is the PL3 Configuration DB definition
//
typedef struct {
   unsigned int       version                      // PL3 database format version
   device_struct      keyboard;                    // Pre-attached keyboard device
   device_struct      screen;                      // Pre-attached screen device
   unsigned int       num_segs;                    // # of segments created for app
   segment_struct     seg[MAX_DSEGS+MAX_MSEGS];
} application_struct;
```

The device_struct type provides the major and minor numbers for devices that have been logically pre-attached by the LPSK during initialization. The segment_struct type provides a structure that allows the SCOS to pass out dseg and mseg information to the application.

The data provided to an application will be a 16-bit selector placed on the stack by the SCOS. The SCOS will then perform an initial outer-ring return to the application. From the point of view of the application, it will appear that execution has returned after making a call, with the selector treated as an output of the call. The selector will appear on the stack as shown in Table 1. The name in the table is informational, and does not represent a named structure.

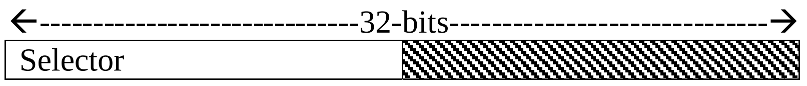
←------------------------------32-bits------------------------------→
Selector

Table 1. Stack Frame for Outer-Ring Return

The application creates a pointer using the selector with an offset of 0. The pointer will point to the memory layout shown in Table 2.

←------------------------------32-bits------------------------------→	
version	
keyboard.major	
keyboard.minor	
screen.major	
screen.minor	
num_segs	
seg[0].selector	▨
seg[0].size	
seg[0].perms	
seg[0].id[0]……………………………………..seg[0].id[3]	
…	
seg[0].id[60]………………………………..…seg[0].id[63]	
…	
seg[95].selector	▨
seg[95].size	
seg[95].perms	
seg[95].id[0]…………………………………..seg[95].id[3]	
…	
seg[95].id[60]…………………….…….…..…seg[95].id[63]	

Table 2. Memory Layout of the PL3 Configuration Database

The dsegs in the configuration database can be uniquely identified by the application by referencing the associated path name in the "id" field. The msegs are identified by the order they were given in the initialization database; the same order given in the initialization database will be the order given in the mseg array. Size alone is not a good identifier because some msegs may be the same size, but shared in different ways (or not at all). The perms element indicates the permission(s) that the application has been assigned for the associated segment, which is either read-only or read-write.

4 Native Call Interfaces

This section describes the native call interfaces provided by the SCOS for applications. When implemented, the calls will be made via a gate interface (e.g., a call gate). The following calls are described in this section:

- scos_flush_segment
- scos_read_next
- scos_read_random
- scos_write_next
- scos_write_random
- scos_shutdown
- scos_powerdown
- scos_login
- scos_who
- scos_logout
- scos_set_password

The following subsections describe the native call interfaces in detail. Note that a conscious decision was made to not have the SCOS validate input arguments for pass-through LPSK functions unless there was a specific SCOS-added feature. Validating inputs would unnecessarily impact performance. The LPSK will make sure security is not violated.

The constants referenced in this section are either defined in LPSK or SCOS header files. For interfaces that return a success code, if the requested action is successful, then the value of NO_ERROR is returned to the caller.

4.1 *scos_flush_segment*

The scos_flush_segment call is used to write the contents of a memory segment to its associated location on secondary storage. The scos_flush_segment call has the lowest implementation priority of all the calls and may not be implemented in Phase 0.

4.1.1 Prototype

unsigned int scos_flush_segment(const selector_type selector);

4.1.2 Inputs

- selector

  The input selector references the segment to be flushed to secondary storage.

4.1.3 Outputs

- Function Result
  The only output is a numerical value that indicates the success or failure of the requested operation.

4.1.4 Processing

1. Call scss_flush_segment using the same inputs and outputs given in scos_flush_segment.
2. Return the function result returned by scss_flush_segment.

4.1.5 Effects

- If successful, the data on secondary storage will be overwritten with known values.
- If an error occurs during the flush, then the data on secondary storage is in an unknown state (i.e., a partial write may occur).

4.1.6 Errors

LPSK_NOT_MAPPED
: This error is returned if the memory object referenced by the input selector was not initialized from a memory object on secondary storage (i.e., the segment is not a dseg).

LPSK_NO_WRITE
: This error is returned if the requesting subject does not have a write permission associated with the dseg to be flushed.

LSPK_WRITE_FAILURE
: This error is returned if an unexpected write error occurred during the flushing of the segment.

4.2 *scos_read_next*

The scos_read_next call is used to read data sequentially from the requested device, such as a keyboard. The call will return no more data in the input buffer than requested by the caller, but it may return less than requested. The actual number of data units returned is provided as an output.

4.2.1 Prototype

```
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);
```

4.2.2 Inputs

- major
  The major device number. It is the number that represents the kind of the device to read from.
- minor
  The minor device number. It is the number that represents the specific instantiation of the major device to be read from.
- num_requested
  The requested number of device-dependent data units (e.g., bytes) to put into the memory location pointed to by the input buffer.
- buffer
  The memory location where a copy of the requested data is to be put.

4.2.3 Outputs

- num_read
  The actual number of data units read and copied into the input buffer. It is possible that the number of units read is smaller than the input num_requested, which will result in no returned error.
- Function Result
  A numerical value that indicates the success or failure of the requested operation.

4.2.4 Processing

1. Call scss_read_next using the same inputs and outputs given in scos_read_next.
2. Return the function result returned by scss_read_next.

4.2.5 Effects

- If data is obtained from the requested device, then the data is copied to the input buffer.

- In the event that an error occurs during the copy operation an error will be returned to the caller, but the state of the input buffer will be unknown. In other words, the buffer may contain only a partially requested read operation.
- If the input buffer provided by the caller points to an area where the requesting subject does not have write permission then a general protection fault will occur when an attempt is made to copy the requested data into the buffer. The effect in Phase 0 will be an ungraceful shutdown of the system.

4.2.6 Errors

LPSK_BAD_MAJOR

This error is returned if the input major is not a valid major device number.

LPSK_BAD_MINOR

This error is returned if the input (major, minor) is not associated with an attached device.

LPSK_NO_DATA

This error is returned if there is no data to read from the device.

LSPK_READ_FAILURE

This error is returned if an unexpected read error occurred while reading from the device.

4.3 scos_read_random

The scos_read_random call is used to read data from the requested device. It is used to read from devices that can be randomly accessed, such as the screen. The call will return no more data in the input buffer than requested by the caller, but it may return less than requested. The actual number of data units returned is provided as an output.

4.3.1 Prototype

```
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);
```

4.3.2 Inputs

- major
  The major device number. It is the number that represents the kind of the device to read from.
- minor
  The minor device number. It is the number that represents the specific instantiation of the major device to be read from.
- offset
  The device-specific location to start the read operation from.
- num_requested
  The requested number of device-dependent data units (e.g., bytes) to read from the device, starting at the input offset.
- buffer
  The memory location where a copy of the requested data is to be put.

4.3.3 Outputs

- num_read
  The actual number of data units read and copied into the input buffer. It is possible that the number of units read is smaller than the input num_requested.
- Function Result
  A numerical value that indicates the success or failure of the requested operation.

4.3.4 Processing

1. Call scss_read_random using the same inputs and outputs given in scos_read_random.
2. Return the function result returned by scss_read_random.

4.3.5 Effects

- If data is obtained from the requested device, then the data is copied to the input buffer.
- In the event that an error occurs during the copy operation an error will be returned to the caller, but the state of the buffer will be unknown. In other words, the buffer may contain only a partially requested read operation.
- If the input buffer provided by the caller points to an area where the requesting subject does not have write permission then a general protection fault will occur when an attempt is made to copy the requested data into the buffer. The effect in Phase 0 will be an ungraceful shutdown of the system.

4.3.6 Errors

LPSK_BAD_MAJOR
: This error is returned if the input major is not a valid major device number.

LPSK_NOT_RANDOM
: This error is returned if the input (major, minor) device cannot be randomly accessed.

LPSK_BAD_MINOR
: This error is returned if the input (major, minor) is not associated with an attached device.

LPSK_BAD_OFFSET
: This error is returned if the input offset was invalid for the given device in its current state.

LSPK_DEVICE_END
: This error is returned if the caller attempts to start a read request from the end of the randomly addressable space.

LSPK_READ_FAILURE
: This error is returned if an unexpected read error occurred while reading from the device.

4.4 *scos_write_next*

The scos_write_next call is used to write data to the requested device. The call will write no more data to the device than requested by the caller, but it may write less than requested. The actual number of data units written is provided as an output.

4.4.1 Prototype

```
unsigned int scos_write_next(
        const unsigned int  major,
        const unsigned int  minor,
        const unsigned int  num_requested,
        const void * const  buffer[1],
              unsigned int  * num_written);
```

4.4.2 Inputs

- major
  The major device number. It is the number that represents the kind of the device to write to.
- minor
  The minor device number. It is the number that represents the specific instantiation of the major device to write to.
- num_requested
  The number of device-dependent data units (e.g., bytes) to write to the device.
- buffer
  The memory location of the data to be written.

4.4.3 Outputs

- num_written
  The actual number of data units written. The num_written may be less than the num_requested.
- Function Result
  A numerical value that indicates the success or failure of the requested operation.

4.4.4 Processing

1. Call scss_write_next using the same inputs and outputs given in scos_write_next.
2. Return the function result returned by scss_write_next.

4.4.5 Effects

- If data is written to the requested device, then the state of the device may have changed, depending on the device.

[1] The use of two const type specifiers means that neither the pointer nor the contents pointed to by the pointer can be modified by the called function.

- In the event that an error occurs during the write operation then an error will be returned to the caller, but the state of the device will be unknown. In other words, a partial write may have occurred.

4.4.6 Errors

LPSK_BAD_MAJOR
: This error is returned if the input major is not a valid major device number.

LSPK_BAD_MINOR
: This error is returned if the input (major, minor) is not associated with an attached device.

LPSK_NO_WRITE
: This error is returned if the calling application does not have write access to the attached device.

LSPK_WRITE_FAILURE
: This error is returned if an unexpected write error occurred while writing to the device.

4.5 *scos_write_random*

The scos_write_random call is used to write data to the requested device. It is used to write to devices that can be randomly accessed, such as the screen. The call will write no more data to the device than requested by the caller, but it may write less than requested. The actual number of data units written is provided as an output.

4.5.1 Prototype

```
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  buffer[2],
              unsigned int  * num_written);
```

4.5.2 Inputs

- major
  The major device number. It is the number that represents the kind of the device to write to.
- minor
  The minor device number. It is the number that represents the specific instantiation of the major device to write to.
- offset
  The device-specific location to start the write operation.
- num_requested
  The number of device-dependent data units (e.g., bytes) to write to the device.
- buffer
  The memory location of the data to be written.

4.5.3 Outputs

- num_written
  The actual number of data units written. The num_written may be less than the input num_requested.
- Function Result
  A numerical value that indicates the success or failure of the requested operation.

4.5.4 Processing

1. Call scss_write_random using the same inputs and outputs given in scos_write_random.
2. Return the function result returned by scss_write_random.

[2] The use of two const's means that neither the pointer nor the contents pointed to by the pointer can be modified by the called function.

4.5.5 Effects

- If data is written to the requested device, then the state of the device may have changed, depending on the device.
- In the event that an error occurs during the write operation then an error will be returned to the caller, but the state of the device will be unknown. In other words, a partial write may have occurred.

4.5.6 Errors

LPSK_BAD_MAJOR	This error is returned if the input major is not a valid major device number.
LSPK_NOT_RANDOM	This error is returned if the input (major, minor) device cannot be randomly accessed.
LPSK_BAD_MINOR	This error is returned if the input (major, minor) is not associated with an attached device.
LPSK_NO_WRITE	This error is returned if the calling application does not have write access to the attached device.
LPSK_BAD_OFFSET	This error is returned if the input offset is invalid for the given device in its current state.
LPSK_DEVICE_END	This error is returned if the caller attempts to start a write operation from the end of the randomly addressable space.
LSPK_WRITE_FAILURE	This error is returned if an unexpected write error occurred while writing to the device.

4.6 *scos_shutdown*

The scos_shutdown call is used to perform a graceful shutdown of the computer. When the termination is complete an optional message is displayed, followed by a system notice advising the user that the shutdown is done. The computer power is kept on. This initial Phase 0 implementation does not support termination signals to non-TML subjects.

4.6.1 Prototype

void scos_shutdown(const char * const[3] message);

4.6.2 Inputs

- message
  A message to display on the screen. The string must be null-terminated and cannot exceed MAX_STOP_MSG bytes

4.6.3 Outputs

None

4.6.4 Processing

1. Call scss_shutdown using the same inputs given in scos_shutdown.

4.6.5 Effects

- All processes are terminated.
- All non-LPSK processing is stopped.
- The state of the screen is minimally altered by the system notice. If the input message is a valid pointer to a non-empty string then the screen state is further altered.
- Because termination signals are not supported in Phase 0, a call to scos_shutdown from one partition will result in processes in other partitions having no opportunity to flush modified segments to disk. On-disk files may then be out of date.

4.6.6 Errors

None

[3] The use of two const's means that neither the pointer nor the contents pointed to by the pointer can be modified by the called function.

4.7 scos_powerdown

The scos_powerdown call is used to perform a graceful power down of the computer. If the computer does not support a software-controlled power down, then the result is the same as a call to scos_shutdown with no message. This initial Phase 0 implementation does not support termination signals to non-TML subjects.

4.7.1 Prototype

void scos_powerdown(void);

4.7.2 Inputs

None

4.7.3 Outputs

None

4.7.4 Processing

1. Call scss_powerdown.

4.7.5 Effects

- All processes are terminated.
- If software controlled power down is supported, then the computer is powered down.
- Even if power down is not supported, all non-LPSK processing is stopped.

4.7.6 Errors

None

4.8 *scos_login*

The scos_login call is used to authenticate the user. It uses a pre-installed password database managed by the SCOS to assist in the authentication. The user-visible consequence of a successful login is application dependent. Only one user may be logged in at a time per SCOS instantiation. Each SCOS instantiation manages its own password database, and no communication between SCOS instantiations is performed. Therefore, if an SCOS authenticates a user for an application running in a partition, the SCOS in a different partition would have no record of that authentication,

4.8.1 Prototype

```
unsigned int scos_login(
        const char * const username,
        const char * const password);
```

4.8.2 Inputs

- username
  The asserted identity. The string must be null-terminated and cannot exceed MAX_USERNAME bytes in length.
- password
  The password to use as proof of the identity. The string must be null-terminated and cannot exceed MAX_PASSWORD bytes in length.

4.8.3 Outputs

- Function result
  A numerical value that indicates the success or failure of the requested operation.

4.8.4 Processing

1. Refer to the SCOS user authentication database to verify that no user is currently logged in. If a user is already logged in then return the SCOS_LOGIN_BUSY error. Otherwise continue to the next step.
2. Verify that the length of the input username is not greater than MAX_USERNAME bytes. If it does exceed the maximum length, then return the SCOS_INVALID_STRING error. Otherwise continue to the next step.
3. Verify that the length of the input password is not greater than MAX_PASSWORD bytes. If it does exceed the maximum length, then return the SCOS_INVALID_STRING error. Otherwise continue to the next step.
4. Obtain the hashed password for the input username by referring to the SCOS password database. If no entry exists for the input username, then return the SCOS_AUTH_FAILURE error. Otherwise continue to the next step.
5. Hash the input password.
6. If the hash generated in the previous step is not equal to the hash obtained from the password database return SCOS_AUTH_FAILURE. Otherwise continue to the next step.

7. Update the SCOS user authentication database to show that the input username is logged in.
8. Return NO_ERROR to the caller.

4.8.5 Effects

- The SCOS user authentication database is updated to reflect that the user associated with the input username has been authenticated (i.e., is considered logged in).

4.8.6 Errors

SCOS_LOGIN_BUSY

> This error is returned if a user is already considered logged in.

SCOS_INVALID_STRING

> This error is returned if either the input username or input password exceeds their maximum length of MAX_USERNAME and MAX_PASSWORD, respectively.

SCOS_AUTH_FAILURE

> This error is returned in one of two cases: 1) The input username does not have a matching entry in the password database; or 2) The hashed input password is not equal to the hashed password stored in the password database for the input username.

4.9 scos_who

The scos_who call is used to query which user (if any) is currently logged in.

4.9.1 Prototype

void scos_who(char * const username);

4.9.2 Inputs

None

4.9.3 Outputs

- username
  The caller passes in this pointer. If a user is logged in then the user name will be copied to this referenced location, followed by a NULL character ('\0'). If no user is currently logged in then an empty string will be copied to the referenced location (i.e., a NULL character in the first byte). To avoid unexpected behavior, the referenced memory location must have at least (MAX_USERNAME + 1) bytes that can be overwritten.

4.9.4 Processing

1. Copy the user name from SCOS authentication database to the output username.
2. Return to the caller.

4.9.5 Effects

- The PL3 memory location referenced by the output username will be modified.

4.9.6 Errors

None

4.10 scos_logout

The scos_logout call is used to request the SCOS to log out the current user. The user-visible consequence of a successful logout is application dependent.

4.10.1 Prototype

void scos_logout(void);

4.10.2 Inputs

None

4.10.3 Outputs

None

4.10.4 Processing

1. Update the SCOS user authentication database to show that no user is currently logged in.
2. Return to the caller.

4.10.5 Effects

- If a user was logged in then the user authentication database is modified to show that no user is logged in.

4.10.6 Errors

None

4.11 scos_set_password

The scos_set_password call is used to change the password for the currently logged in user. It is up to the calling application to ensure the user has provided the intended password (e.g., by requiring the user to enter the password twice) and that the person providing the new password is the actual user (e.g., by requiring a re-entry of the current password). Phase 0 does not enforce password selection policies. If the scos_flush_segment call is not implemented in Phase 0, then the change will not persist across a reboot of the computer.

4.11.1 Prototype

```
unsigned int scos_set_password( const char * const password );
```

4.11.2 Inputs

- password
  The new password for the currently logged in user. The string must be null-terminated and cannot exceed MAX_PASSWORD bytes in length.

4.11.3 Outputs

- Function result
  A numerical value that indicates the success or failure of the requested operation.

4.11.4 Processing

1. Verify that a user is currently logged in by referring to the SCOS user authentication database. If no user is currently logged in, then return the SCOS_NO_LOGIN error.
2. Verify that the length of the input password is not greater than MAX_PASSWORD bytes. If it does exceed the maximum length, then return the SCOS_INVALID_STRING error.
3. Hash the input password.
4. Replace the hashed password in the SCOS password database that corresponds to the logged in user (as noted in the SCOS authentication database).
5. Call scss_flush_segment to update the on-disk copy of the password database.
6. Return the function result returned by scss_flush_segment to the caller.

4.11.5 Effects

- If successful, then the on-disk copy of the SCOS password database is updated to reflect a change in a user's password.

4.11.6 Errors

SCOS_NO_LOGIN

This error is returned if there is no user currently logged in.

SCOS_INVALID_STRING

This error is returned if the input password exceeds the maximum length of MAX_PASSWORD.

LSPK_WRITE_FAILURE

This error is returned if an unexpected write error occurred during the flushing of the password database.

5 SP Emulation Call Interfaces

The SCOS call interfaces that support an SP emulation were originally specified in a separate document [4]. However, some changes were made to the interfaces after the document was prepared. Therefore, the current interfaces are provided below, with the "scos_" prefix added to the calls. These "scos_" calls are pass-through calls to the equivalent "scss_" calls.

```
/* 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 long 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 scos_SPHW_DeviceRootKey_Set (unsigned int sel, const gpreg_t rs1, const
        gpreg_t rs2);
SPFault scos_SPHW_DeviceRootKey_Lock(void);
SPFault scos_SPHW_DeviceRootKey_Derive(void);
SPFault scos_SPHW_StorageRootHash_Get(void);
SPFault scos_SPHW_StorageRootHash_Set(void);
SPFault scos_SPHW_CEMBuffer_Set (unsigned int sel, gpreg_t *rd);
SPFault scos_SPHW_CEMBuffer_Get (unsigned int sel, const gpreg_t rs1,
        const gpreg_t rs2);
SPFault scos_SPHW_BeginCEM_auth (void);
SPFault scos_SPHW_EndCEM_auth (void);
SPFault scos_SPHW_SecureArea_Add (void *addr, size_t size);
SPFault scos_SPHW_SecureArea_Remove (const unsigned int opts, void *addr);
SPFault scos_SPHW_SecureArea_Store (void *addr);
SPFault scos_SPHW_SecureArea_Load (void *addr);
```

6 End Notes

This section contains notes for future SCOS phases.

If a future phase allows the SCOS to support more than one application, then the processing for call interfaces that accept pointers as inputs and outputs must be expanded. In such situations, the SCOS must verify that the pointers belong to the calling application, and not to any other application supported by an instantiation of the SCOS.

If a future phase allows the SCOS to support more than one application, some thought must be given to a policy regarding the attachment of devices. If application A attaches a device via an SCOS call, then application B should not be able to use it. Therefore, the SCOS must ensure that when an application makes a call that includes the passing of a major/minor pair, that the device is attached by the application before allowing the device operation to occur.

Much of Section 2 “PL3 Initialization Database” should be moved to a specification for the configuration vector tool.

References

[1] Dwoskin, Jeffrey, Lee, Ruby, “Hardware-rooted Trust for Secure Key Management and Transient Trust”, ACM CCS, 2007.

[2] Clark, Paul C., Irvine, Cynthia E., Levin, Timothy E., Nguyen, Thuy D., Vidas, Timothy M., SecureCore Software Architecture: Trusted Path Application (TPA) Requirements, NPS-CS-07-001, Naval Postgraduate School, December 2007.

[3] GNU Grub, http://www.gnu.org/software/grub/

[4] Dwoskin, Jeffrey, Lee, Ruby, “SP Processor Architecture Reference and SP Emulation Module & SP TSM Interface (Draft)”, Version 0.5, September 30, 2007.

Initial Distribution List

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library, Code 013 2
   Naval Postgraduate School
   Monterey, CA 93943-5100

3. Research Office, Code 09 1
   Naval Postgraduate School
   Monterey, CA 93943-5138

4. Karl Levitt 1
   National Science Foundation
   4201 Wilson Blvd.
   Arlington, VA 22230

5. Lee Badger 1
   DARPA
   3701 Fairfax Drive
   Arlington, VA 22203

6. Paul C. Clark 1
   Code CS/Cp
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5118

7. Cynthia E. Irvine 2
   Code CS/Ic
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5118

8. Timothy E. Levin 2
   Code CS/Tl
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5118

9.	Thuy D. Nguyen Code CS/Tn Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5118	2
10.	Timothy M. Vidas Code CS Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5118	2

[THIS PAGE IS INTENTIONALLY BLANK]