THESIS

LEAST PRIVILEGE SEPARATION KERNEL STORAGE HIERARCHY PROTOTYPE FOR THE TRUSTED COMPUTING EXEMPLAR PROJECT

by

Jonathan Guillen

June 2010

Thesis Co-Advisors: Cynthia E. Irvine
Paul C. Clark

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2010	**3. REPORT TYPE AND DATES COVERED** Master's Thesis
4. TITLE AND SUBTITLE Least Privilege Separation Kernel Storage Hierarchy Prototype for the Trusted Computing Exemplar Project		**5. FUNDING NUMBERS**
6. AUTHOR(S) Jonathan M. Guillen		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ________________.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The Least Privilege Separation Kernel (LPSK) is part of the Trusted Computing Exemplar (TCX) project. Separation kernels may be used to partition resources in support of the enforcement of mandatory security policies. The LPSK provides services that allow each subject to access resources configured as part of its domain..To ensure permanence of information the LPSK requires a storage hierarchy for its data resources.

This thesis describes the design for a LPSK storage hierarchy based on existing LPSK requirements. The design was implemented in a Linux environment to produce a storage hierarchy prototype. Implementation of the prototype proceeded in keeping with principles for developmental security which include minimization, modularity, and hierarchical dependencies. The LPSK storage hierarchy external interfaces belong in three distinct categories: The configuration interfaces are used to construct the storage hierarchy and its contents in a non-LPSK context, initialization interfaces associate data segment handles with data segments that are exported to LPSK subjects, and runtime interfaces support the reading and writing to secondary storage data segments exported to non-LPSK subjects. Testing showed that storage hierarchy interfaces behaved according to specification. This study shows that a storage hierarchy prototype can be designed and implemented based on the LPSK functional specification.

14. SUBJECT TERMS Trustworthy systems, separation kernels, secondary storage, storage hierarchy			**15. NUMBER OF PAGES** 165
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

LEAST PRIVILEGE SEPARATION KERNEL STORAGE HIERARCHY PROTOTYPE FOR THE TRUSTED COMPUTING EXEMPLAR PROJECT

Jonathan M. Guillen
Civilian, Naval Postgraduate School
B.S., Computer Science, Western Oregon University, 2008

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2010

Author: Jonathan Michael Guillen

Approved by: Cynthia E. Irvine
Thesis Advisor

Paul C. Clark
Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Least Privilege Separation Kernel (LPSK) is part of the Trusted Computing Exemplar (TCX) project. Separation kernels may be used to partition resources in support of the enforcement of mandatory security policies. The LPSK provides services that allow each subject to access resources configured as part of its domain. To ensure permanence of information the LPSK requires a storage hierarchy for its data resources.

This thesis describes the design for a LPSK storage hierarchy based on existing LPSK requirements. The design was implemented in a Linux environment to produce a storage hierarchy prototype. Implementation of the prototype proceeded in keeping with principles for developmental security which include minimization, modularity, and hierarchical dependencies. The LPSK storage hierarchy external interfaces belong in three distinct categories: The configuration interfaces are used to construct the storage hierarchy and its contents in a non-LPSK context, initialization interfaces associate data segment handles with data segments that are exported to LPSK subjects, and runtime interfaces support the reading and writing to secondary storage data segments exported to non-LPSK subjects. Testing showed that storage hierarchy interfaces behaved according to specification. This study shows that a storage hierarchy prototype can be designed and implemented based on the LPSK functional specification.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

CC	Common Criteria
CISR	Center for Information Systems Studies and Research
EAL	Evaluation Assurance Level
GCC	GNU Compiler Collection
GTNP	Gemini Trusted Network Processor
HAL	Hardware Abstraction Layer
LPSK	Least Privilege Separation Kernel
MAC	Message Authentication Code
PFIP	Partitioned Information Flow Policy
SAT	Segment Allocation Table
SKPP	Separation Kernel Protection Profile
TCX	Trusted Computing Exemplar

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Dr. Cynthia Irvine, and co-advisor, Paul Clark, for their tremendous support, encouragement, and most significantly, their patience with me throughout the thesis development process. I would like to thank David Shifflett for several technical insights he provided. I would also like to thank Valerie Linhoff for her support of my studies at the Naval Postgraduate School.

Finally, I would like to thank my parents for encouraging and supporting my academic pursuits.

This material is based upon work supported by the National Science Foundation, under grand No. DUE-0414012. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author, and do not necessary reflect the views of the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

The size and complexity of modern computing systems, and governmental and private organization's reliance on these systems to process sensitive information, has increased the attention paid to the threat of system exploitation and unintended information flow. Users rely on system security mechanisms to enforce policies regarding the flow of information between various system components. A system's security policy however is not always equivalent to the actual security policy enforced by the system. The certainty of a system's ability to enforce a security policy can only be measured by the analysis of policy-enforcing mechanisms. A high level of certainty, or assurance, that a system will behave as configured increases confidence in that system's ability to correctly process sensitive information.

The goal of the Trusted Computing Exemplar (TCX) project is to develop an example of a high assurance computing platform. The core operating system, or kernel, of the TCX computing platform will provide a high level of assurance that only explicitly defined information flow can occur between entities hosted on the platform. The Least Privilege Separation Kernel (LPSK) contains the mechanisms for separating system subjects and resources into *partitions*, as if they were located in distributed systems, and controlling the flow of information between these partitions. This information flow policy is part of the configuration of an LPSK system.

Operating systems need file systems to store data over long periods of time. So does the LPSK. The LPSK provides long-term persistence of information for its hosted subjects through the LPSK storage hierarchy. The motivation of this thesis is to develop a storage hierarchy prototype that meets the requirements of the TCX project and the functional requirements of the LPSK. The storage hierarchy prototype is an LPSK module that facilitates the secondary storage of *data segments* in a tree-like structure.

While similar to conventional file systems in many ways, the storage hierarchy differs from these systems in that it is designed to intentionally avoid the potential of information flow through a resource exhaustion covert channel.

B. PURPOSE OF STUDY

The objective of this research was to analyze the LPSK storage hierarchy requirements, map these requirements to a storage hierarchy design, and then implement this design as a prototype LPSK module. An interface specification document was authored to define the boundary between the storage hierarchy module and the LPSK and LPSK configuration tools. The module was tested to ensure it behaved according to the interface specification. This study contributes original work to the TCX project and furthers its goals of developing an openly available example of a high assurance computing platform.

C. THESIS ORGANIZATION

This thesis is organized as a series of chapters. Chapter I describes the motivation and purpose of this thesis. Chapter II provides background information on the TCX project, separation kernels and the LPSK, file system organization, and the LPSK storage hierarchy. Chapter III contains the storage hierarchy requirements, interface and data structure designs, and implementation details. This chapter makes the distinction between three types of storage hierarchy external interfaces. Chapter IV documents storage hierarchy external interface test results. Functional testing was performed on all interfaces in conjunction with negative testing for error cases. Chapter V concludes this work with a discussion of the study's results, lessons learned, and suggestions for future work.

II. BACKGROUND

This chapter serves to provide the reader with basic understanding of the Least Privilege Separation Kernel of the Trusted Computing Exemplar project and the LPSK Storage Hierarchy specifically. The concepts of trusted computing and separation kernels are introduced in the first section. The second section introduces the Storage Hierarchy as an analog of file systems and discusses file system concepts as they relate to the LPSK storage hierarchy.

A. TRUSTED COMPUTING EXEMPLAR PROJECT

Public demand for computer software technology, and the rapid development of technology to satisfy this demand, has resulted in insufficient consideration given to the trustworthiness of systems intended to protect both code and data. There are no recent openly available examples of highly trustworthy software products that show how such systems are built [1]. The purpose of the TCX project is to address this lack of trusted computing technology by developing open and documented examples of software systems that can be verified and proven to provide high assurance. High assurance is achieved by showing that systems operate correctly and according to specification. Output of the TCX project will satisfy the demand for knowledge in the area of trusted computing. The four main activities of the TCX project follow are:

- Creation of a prototype framework for rapid high assurance system development;
- Development of a reference-implementation trusted computing component;
- Evaluation of the component for high assurance; and
- Open dissemination of deliverables related to the first three activities [1].

The following three sections discuss trusted computing, separation kernels, and the TCX Least Privilege Separation Kernel.

1. Trusted Computing

Trusted Computing is an approach to computer architecture and software development and evaluation that considers both the risks of system penetration by frontal attack and system subversion. The methods employed to show that both categories of risk are properly addressed will also show that the system in question operates correctly and can thus be trusted.

Frontal attacks are those that exploit behaviors in a system that were inadvertently included by its developers. Frontal attacks rely on the exploitation of either documented or undocumented flaws in a system. Subversion is the purposeful placement of an artifice, or trap door, into a system during design or implementation that circumvents normal system controls when instructed to do so by specific stimulus [2]. Subversion can occur at any time during a system's life cycle and can be designed to completely bypass all system controls meant to protect the system's integrity. Ensuring that either case of attack is not possible requires a system to be implemented with no exploitable flaws and without subversive artifices.

Modern computer systems are often comprised of hundreds of thousands, or even tens of millions, of lines of code. It is infeasible to analyze such large and complex systems by source code inspection or security test and evaluation to determine a lack of exploitable flaws and malicious artifices [3][4]. Furthermore, subversion of a system can occur in its earliest development phases, confounding future attempts at detection. Trusted Computing methodologies are intended to prevent subversion by requiring exact and unambiguous specification of all system databases and functions from the very first design phases of the system. When combined with rigorous configuration management, this increases confidence in the integrity of the system from inception through retirement.

Formal methods are used to show that the security policy of the system enforced by its security mechanisms is mathematically correct. A system's security policy represents all permissible interactions between subjects or processes and data. Since these permissible interactions are potential targets of subversion, the security mechanisms that enforce the security policy of the system must be verifiable against some standard criterion to show they work correctly. The Common Criteria (CC) supports the creation of a common set of requirements and evaluation techniques to express some measure of confidence in this verification of security mechanisms [5].

The CC was developed by several cooperating countries to provide a recognized standard for providing security guidance to vendors, and a framework to evaluate system controls and security mechanisms [5]. A protection profile is developed to require a particular level of assurance, with higher evaluation assurance levels (EALs) requiring more stringent and exacting requirements. For instance, successful EAL1 validation provides confidence a system will operate correctly where security threats are of little concern, while EAL7 validation provides confidence that the security policies will be correctly enforced even in the face of major threats. A system that is evaluated at EAL7 will have been developed with both frontal attacks and subversion risks fully considered and guarded against.

Trusted Computing attempts to prevent successful frontal attack and system subversions by addressing the origination of these risks in planning and implementation. By formally modeling and proving the correctness of a system's design, and validating its implementation against a set of standards such as the CC, the trustworthiness of a system can be established.

2. Separation Kernels

The kernel is the core of an operating system and has direct and complete control over all of the systems resources, including the operating system, and user applications and data. The interaction of applications and users by way of data flow in the system is mediated by controls in the kernel. Without assurance provided for the correctness of these kernel controls, no assertions can be made about the interactions and data flows they are meant to control.

Security kernels attempt to provide assurance by consolidating their basic controls and security mechanisms into a minimal, verifiably correct core, or kernel. The interactions and data flows of the system are then mediated from this trusted core, which becomes the enforcer of a system-wide security policy [6][7]. This approach is complicated with the introduction of “trusted processes” to the system [8]. These processes are less constrained by the mechanisms of the security kernel in order to carry out necessary tasks that are otherwise forbidden by the system’s security policy: for example, re-grading the classification level of information in a multilevel system. The verification of a system that uses trusted processes must now extend outside the security kernel to those processes, contrary to the original intent of these kernels [8].

In 1981, Rushby suggested a new class of security kernels called separation kernels [8]. He observes that no data flow can occur in a physically separated system of individual computers by virtue of their disconnectedness. The data flow policy of the overall system is determined by how the individual computers are physically connected by communication channels. Individual computers in the distributed system may not influence the operation of other computers unless communication is authorized by the data flow policy. A separation kernel emulates the properties of this distributed system in one physical system by creating multiple, virtual execution environments, and enforcing a data flow policy between them [8].

Separation kernels separate the resources of a system (such as subjects and data) into partitions. Subjects in one partition may not interact with data in another partition unless this interaction has been explicitly allowed by the data flow policy enforced by the separation kernel. The data flow policy of the system is defined by the configuration of the partitions, their contents, and the communication channels between them.

The Information Assurance Directorate of the United States Government published “The U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness.” This document defines the Separation Kernel Protection Profile (SKPP); the requirements for a high assurance separation kernel [9]. The TCX LPSK is being built to comply with the SKPP, and is targeted for evaluation at EAL7.

Figure 1 illustrates a simple separation kernel configuration which contains three partitions, three subjects, and a number of resources distributed amongst them. Arrows represent interaction or the flow of information to or from subjects. An arrow pointing towards a subject denotes that subject's ability to read from the source of the arrow. An arrow pointing from a subject denotes the ability to write. A double-headed arrow shows read/write capability. The SKPP refers to a partitioned information flow policy (PFIP) to describe data flow between subjects and resources of the same or different partitions. This allows for more granular data flow control than Rushby's general inter-partition communication concept [9]. In this example, Subject 2 located in Partition A is capable of writing to a resource in Partition B, which is also read/writable by Subject 3. Subject 3, however, has not been allowed to communicate with other resources in his partition. This flow of data that occurs between partitions, and between subjects and resources within partitions, is explicitly declared in the configuration data of the separation kernel.

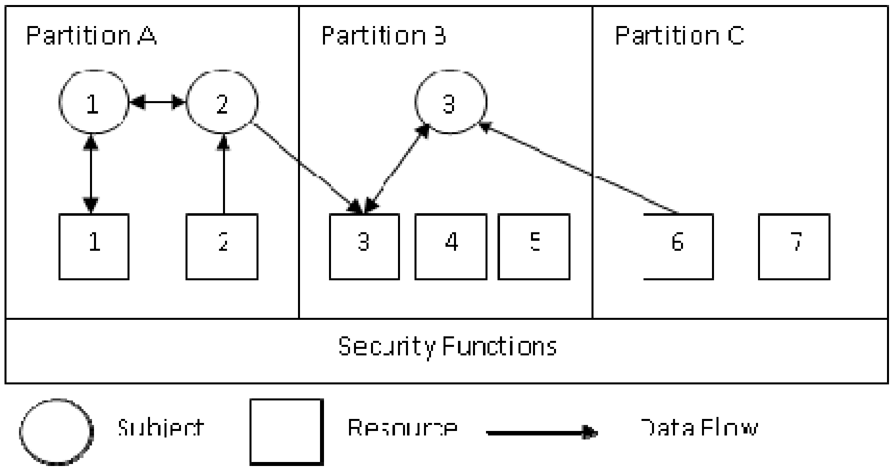


Figure 1. Separation kernel configuration example from SKPP document [9].

The TCX implementation of a SKPP-compliant separation kernel, the LPSK, is discussed in the next section.

3. TCX Least Privilege Separation Kernel

The TCX LPSK is a separation kernel implementation that is compliant with the SKPP [10]. It enforces two data flow policies: partition-to-partition and subject-to-resource. The intersection of these two policies produces a correct SKPP PIFP. The subject-to-resource policy determines what resources a subject may read from or write to. Between partitions, the partition-to-partition policy determines what resources a subject located in another partition may interact with. As required by the SKPP, the LPSK is configured prior to system operation by way of a configuration vector that defines the partitioned subjects and resources, and PIFP of the system [9].

One type of resource made available to subjects in an LPSK system is a segment. Segments represent blocks of data that subjects may read from or write to during the course of their execution. A segment that is ephemeral and exists only in primary memory is referred to as an "mseg" while a segment that is stored long-term across multiple system power cycles is called a data segment, or "dseg." The TCX LPSK Storage Hierarchy, which provides mechanisms for storing and retrieving data segments, is discussed in the next section.

B. STORAGE HEIRARCHY

The applications that run on the LPSK require several basic system services regarding the maintenance and persistence of their data. First, they must be able to use portions of system primary memory to perform their operations, and second they may save the contents of their primary storage to a secondary storage area for later retrieval. Applications may also require certain data be available at the beginning of their execution, e.g., when an LPSK system has just started. This data must be available from some secondary store that is present before system startup.

These expectations are common on many modern operating systems, and have traditionally been served by a memory manager and file system. Memory managers allow applications to reserve portions of system memory while preventing unwanted modification of protected or otherwise unshared memory. File systems store application and system data on some medium, and maintain data structures necessary to track the locations of various data segments. The LPSK Storage Hierarchy subsystem is available to the LPSK memory manager for the storage and retrieval of data segments.

The following sections discuss traditional file systems as secondary storage managers, the LPSK Storage Hierarchy as a specialized subsystem that supports the LPSK mission of data-domain separation, and the concept of data segments and how they are stored.

1. File Systems

Long-term storage of application data in primary storage (main system memory) is not possible due to the volatility of memory. Random-access memory (RAM) maintains its state by applying electrical current to a series of transistors (SRAM) or transistors and capacitors (DRAM). When a system is turned off and the electrical current is no longer applied, any information stored in memory is rapidly lost. The cost of transistor-based storage is also a concern. While much faster in terms of operations per second, transistor-based storage is much more expensive than non-volatile storage. Secondary storage provides long-term, stable storage of data that persists across system power states. This has traditionally been accomplished in contemporary computer systems by use of magnetic hard disk drives, though there is a slow trend toward large-scale solid-state drives. Generally, a file system is responsible for maintaining data on a non-volatile storage medium in a way that makes files accessible to the rest of the system.

The architecture of various storage media has often influenced the design of contemporary file systems. For instance, the smallest unit of data a file system presents to a client to interact with is derived from the way hard disk drives store binary data on their magnetic platters. Even though emerging solid-state drive technology does not share the mechanical limitations of traditional hard disk drives, the conventions developed around the use of hard disk drives are still generally applied to the new class of solid-state drives.

Hard disk drives store binary data on rotating magnetic platters, or disks. The surface of the platters is partitioned into many disk sectors. A sector is read or written in a single atomic operation. Each sector traditionally stores 512 bytes of data, although hard disk drive vendors have agreed to increase sector sizes to a standard of four kilobytes in the near future [11]. The hard disk drive controller—the integrated circuits that facilitate communication between the hard disk drive and an overall computer system—presents the collection of physical sectors as a linear span of addressable blocks [12]. These logical blocks are mapped to physical sectors by the hard disk drive controller. File systems use this logical view of hard disk drives to store files.

A file is a bounded sequence of data bytes that is meaningful to the system or applications that run on a system. A file system is a collection of files and the data necessary to determine the boundaries between files on the medium in which they are stored [13]. A file system’s locater information, or meta-data (data about the data stored in the file system), maintains the actual location of files by address on the medium, addresses available for allocation to files, and the logical structure of a file hierarchy. File systems can provide a logical model of a file hierarchy by identifying the location of files within a tree structure comprised of directories and sub-directories.

File system directories are special interpretively accessed objects that contain information about their direct descendents in the logical hierarchy. This includes both location and metadata information. File systems that maintain only one directory are considered to be flat or single layer file systems, while hierarchical file systems are those that allow the creation of sub-directories as children of the primary directory [13]. In either case, applications that utilize a file system to store data need not be aware of the physical layout of data bytes across the storage medium, but can instead rely on a naming convention to identify and reference files.

The size of files is measured in bytes. Files can be expected to grow or shrink in size as they are modified. They can also be removed entirely. Also new files may be created. In order to accommodate dynamic restructuring of its data, file systems partition their storage media into individual blocks of some byte length and maintain a list referencing every block the file system is responsible for [13]. This partitioning allows a file system to reduce the amount of memory required to address all of its space, while allowing files to be allocated across consecutive or nonconsecutive free blocks using group chaining. Even though a file might not be located on consecutive blocks of the medium due to fragmentation, the file system can follow the linked list produced by group chaining to find each block that comprises the file.

Figure 2 demonstrates the allocation of a file named “myfile” across a series of blocks in a file system. Omega, **Ω**, denotes the end of a group chain of blocks, while an empty block means that block has not been allocated to a file and is available for allocation. Any other value represents the address of the next block in the group chain the file system should read from for the next part of a file. The directory mentioned previously resides at the first block and contains the single entry for “myfile”. The directory shows that the first block of the group chain that contains “myfile” is at address 1 in the overall group of blocks. The entry at block 1 tells the file system that the next block of “myfile” is located after it at block 2. Following the linked list produced by the group chaining, the file system learns that the last block of the file is located at block 5. The fact that block 3 is empty suggests that it was previously allocated to a file that is no longer present in the file system. In the simplest file allocation scheme, if a new file is added to this example layout, it would be allocated block 3 for the first part of the file. If any successive blocks were necessary to store the file, they would be allocated at block 6 and sequentially thereafter.

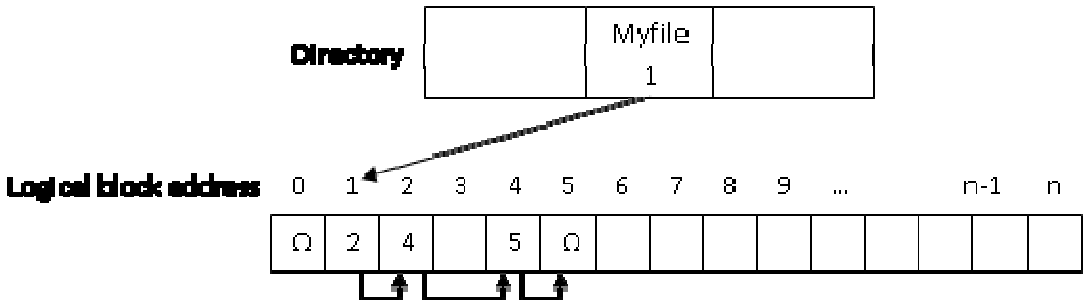


Figure 2. Example of file allocation across a logical file system block set

File systems store system and application data on non-volatile (or volatile) media. They abstract the complexity of organizing files on a storage medium and provide a simple hierarchical structure of files to users and applications. The following section will show that the LPSK Storage Hierarchy is a specialized file system that provides similar functionality while adhering to design specifications of the LPSK and the TCX project.

2. LPSK Storage Hierarchy as a File System

The LPSK Storage Hierarchy (hereafter referred to as the storage hierarchy) provides secondary storage by arranging data into a hierarchical set of data segments. A data segment is an abstraction for data storage and is used as a container for a subject’s data in primary memory and secondary storage [14]. During the operation of the LPSK system, the file system retrieves data segments from the storage hierarchy to load into a subject’s primary memory space, or flushes segments located in primary memory into the appropriate secondary storage segment. In this mode of operation, the storage hierarchy is analogous to a traditional file system, where data segments are similar to files and the overall storage hierarchy is analogous to a file system. However, there are some important distinctions.

Since the resources and data to which a subject has access are statically declared as part of the system’s configuration, no new secondary storage data segments can be created during system operation. Likewise, data segments cannot be destroyed during operation. The dimensions of data segments in terms of their length in bytes are also declared as part of the configuration of the system. These three considerations imply a storage hierarchy whose structure and dimensions are static.

The storage hierarchy organizes data segments in a tree structure beginning with a notional root node, ‘/’, that represents the root directory. Each node of the hierarchy may have one or more child nodes, up to a maximum number of children defined in the configuration of the system. Node entry numbers (the name of the node that differentiates it from its siblings) may be any number between zero and the maximum number of children minus one as mentioned before. Child nodes of the same parent may not share the same label. Concatenating a child node’s label with its parent’s label separated by forward slashes produces the absolute path to that node from the root node. In our interpretation of the storage hierarchy, a child node is either a data segment or a directory node. Data segment nodes are leaf nodes that correspond to secondary storage data segments while directory nodes correspond to internal nodes within the tree that may contain additional children nodes.

Figure 3 shows a logical model of a storage hierarchy which contains six secondary storage data segments and four directory nodes. The arrows show the lineage of parent nodes.

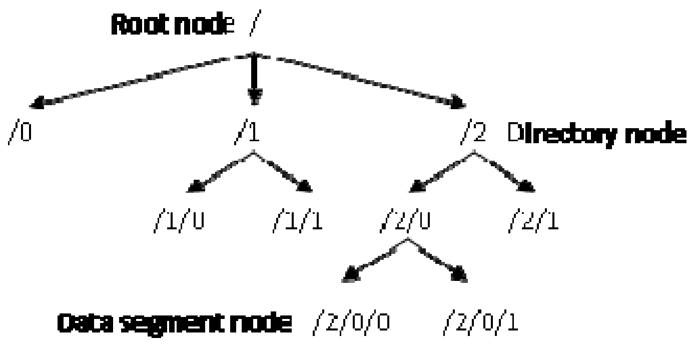


Figure 3. Example of a notational storage hierarchy tree structure

During configuration of an LPSK system, the exact structure of the storage hierarchy and dimensions of data segments are defined, and the contents of the secondary storage data segments are pre-loaded as necessary. The LPSK ensures that only subjects that are configured to access the corresponding data segment in primary memory, as defined by the LPSK system configuration, may access that secondary storage data segment in the storage hierarchy during run-time [14].

Accessibility to secondary storage data segments is provided by the three operations subjects may perform on data segments; swap in, swap out, and flush. A subject's primary memory space is limited by the configuration of the system and indirectly by the physical resources provided by the hardware on which the system runs. Because of potential limited space, subjects may need to move their data segments out of primary memory and into secondary storage, followed by the reverse. Swapping a data segment into a subject's primary memory space loads the contents of the corresponding secondary storage data segment. Swapping out a data segment overwrites the contents of the secondary storage data segment with the contents of the data segment in primary memory and frees that primary memory for the swapping in of other data segments associated with that process. Flushing a data segment overwrites the contents of a secondary storage data segment while keeping the corresponding data segment intact in primary memory.

After an LPSK platform is configured and the LPSK is started, the LPSK initialization routine reads the configuration data and imports the specified data segments into primary memory. The initialization routine manages this importation by passing requests to the storage hierarchy. The storage hierarchy constructs a data structure in main memory that associates a data segment handle with the absolute path of a secondary storage data segment. When the kernel or its subjects need to perform an action on an imported segment, the handle is passed to the storage hierarchy as an identifier of the segment to be acted upon.

This section has demonstrated that the LPSK storage hierarchy is a specialized file system that facilitates the secondary storage of LPSK data segments. The concept of a data segment in primary memory or secondary storage has been introduced along with the tree-like structure of the storage hierarchy. The next chapter of this thesis describes the requirements, technical design, implementation methodology, and implementation of the storage hierarchy.

III. DESIGN AND IMPLEMENTATION

This chapter is separated into four sections: requirements, design, implementation methodology, and implementation. The requirements for the storage hierarchy prototype are described first, followed by a description of how the requirements are met by the design of the storage hierarchy prototype's data structures and interfaces. The third section describes the development environment and tools used to implement the storage hierarchy prototype. The last section describes the implementation of the storage hierarchy interfaces.

A. REQUIREMENTS

The requirements for the storage hierarchy prototype are derived from two sources: a functional specification and a collection of general platform-related requirements. A functional specification of the storage hierarchy is located in section 14.5.1 of the TCX LPSK Product Functional Specification (see Appendix) and includes a list of twelve requirements [14]. These twelve requirements are identified and discussed in the next section, followed by a section addressing an additional set of general requirements that inform the design of the storage hierarchy prototype developed in this thesis.

1. Storage Hierarchy Functional Specification

The TCX LPSK Product Functional Specification details the functional requirements of the LPSK. Section 14.5.1 of this specification lists twelve requirements for the organization of secondary storage data segments (hereafter referred to as data segments). The following list paraphrases these requirements:

1. Data segments are organized in a tree-like hierarchy, where nodes of the tree represent data segments.
2. The root of the tree is a node named '/' and is not used to represent a data segment.

3. Nodes shall have any number of child nodes up to a configured maximum defined when the secondary storage media is formatted.

4. Nodes are identified by a name consisting of any number from zero up to some configured maximum minus one.

5. The name of a data segment is the list of node names from the root node to the specified data segment.

6. Data segments are limited in size to 2^{32} bytes (4 GB).

7. The "/0" node and its children are reserved for internal use by the LPSK.

8. Only software that runs in a non-LPSK context may create data segments in the storage hierarchy.

9. A message authentication code is generated and saved for a data segment whenever it has been created by configuration software or when a data segment in primary memory is flushed to the corresponding data segment in secondary storage during LPSK runtime.

10. The integrity of data segments is verified by the LPSK whenever a data segment is swapped into primary memory from secondary storage. Failure in verification will result in some action according to the context of the system when verification is attempted.

11. The LPSK Initializer provides data segment selectors to subjects that have been configured to access those data segments.

12. During runtime, subjects shall use selectors supplied by the LPSK to perform actions on data segments.

The term "selectors" refers to a special token or handle a subject uses to identify which data segment the subject wishes to have some action performed on. The *LPSK Initializer* is an LPSK module that is responsible in part for creating and supplying the appropriate selectors to the appropriate subjects, based on the configuration of the LPSK system.

The first five requirements taken together prescribe the tree-like structure of the storage hierarchy discussed in Chapter II. In that discussion, a node of the tree is described as either a data segment node or directory node. Early design decisions resulted in this unnecessary and unspecified distinction in a node's type. This exclusive treatment of nodes is imposed by the model of the storage hierarchy developed in this thesis and is not a restriction derived from the listed requirements. Eliminating this distinction would allow an inner node (a node that has child nodes) to be a data segment. In the file system analogy, this lack of distinction is equivalent to allowing the creation of directories and subdirectories as files that may also contain files. The storage hierarchy model described in this thesis does not allow the overloading of nodes—resulting in nodes that are both data segment nodes and directory nodes—to avoid cognitive dissonance arising as a result of the storage hierarchy to file system analogy. These five requirements and the overall structure of the storage hierarchy are achieved through the creation of directory table data structures, which are discussed in detail in a subsequent section.

The sixth listed requirement relates to the storage of a data segment's size in bytes as an unsigned 32-bit integer value. The seventh listed requirement designates a special sub-tree of the storage hierarchy that contains data segments used internally by the LPSK. These segments are not made available to external subjects (subjects executing in LPSK partitions). The storage hierarchy is not responsible for enforcing this policy however, and relies on the LPSK initialization process to restrict selectors for internal use from use by external subjects.

The eighth listed requirement, that only software that does not run as part of the LPSK system may create data segments, leads to the need for an offline storage hierarchy configuration tool. While such a tool is outside the scope of this thesis, the storage hierarchy prototype must include documented external interfaces to allow such a tool to modify the structure and contents of an LPSK storage hierarchy. These configuration interfaces should be executable on the same operating system the configuration tool runs on, such as the open source Linux operating system. These external interfaces are identified in Section B of this chapter.

The ninth and tenth listed requirements call for the creation and maintenance of message authentication codes (MAC) for each data segment in the storage hierarchy. A MAC is generated based on the contents of a data segment and a MAC key known by the LPSK, and is stored in the metadata associated with the data segment. The MAC for a given data segment is updated every time the contents of that data segment changes in a permissible context, e.g., during configuration or when a data segment is swapped out to the storage hierarchy during LPSK system execution. If the contents of a data segment are modified in some other context, such that a new valid MAC is not generated, the LPSK will not permit that data segment to be read into primary memory.

The last two requirements prescribe the process of providing and using data segment selectors. The LPSK provides subjects zero or more selectors based on the configuration of the system. In addition to memory management of segment data, subjects may use these selectors as the target of *swap in*, *swap out*, and *flush* system calls. These requirements are achieved during initialization of an LPSK system by creating a database that maps selectors to data segments, and providing storage hierarchy runtime interfaces that accept selectors as parameters. These interfaces and the selector to data segment database are described in Section B of this chapter.

2. Additional Storage Hierarchy Requirements

All software developed for the TCX LPSK project must adhere to a set of standards described by the Software Development Standards document, a product of the Center for Information Systems Security Studies and Research (CISR). This document specifies that an ANSI-C compliant programming language is to be used when programming kernel code [15]. Any storage hierarchy related code that is executed during LPSK initialization or runtime must therefore be developed with the C programming language.

The LPSK binary (the executable LPSK machine code) must be compiled by the Open Watcom C compiler [16]. This compiler was chosen over other open source compilers—such as the GNU Compiler Collection [17]—because it supports a large memory model [18]. Storage hierarchy configuration interfaces, however, are not included in the LPSK binary due to previously enumerated requirements, and are not necessarily restricted to compilation by the Open Watcom compiler.

The storage hierarchy will interact with its storage medium via a hardware abstraction layer (HAL). A HAL hides a particular device's architecture and presents a set of interfaces common to the overall class of devices, such as hard disk drives. This is a necessary requirement as no hard disk device drivers have been developed for the LPSK at the time of this writing. Testing the storage hierarchy prototype required a special construct located in a test system's memory or secondary storage that mimics an LPSK storage hierarchy's storage medium. A standardized HAL makes it possible for the storage hierarchy prototype to communicate with both the special testing construct and future LPSK hard disk driver software.

B. DESIGN

The design of the storage hierarchy prototype attempts to satisfy the requirements identified in the previous section. Golden and Pechura's description of a hierarchical file system is used as the basis for the design of the storage hierarchy's internal structure [11]. This section describes both the internal structure of storage hierarchy constructs and the various interfaces used to interact with the storage hierarchy.

Figure 4 provides an overview of the users of storage hierarchy interfaces and how those interfaces are organized in relation to the storage medium. The LPSK makes use of both initialization and runtime interfaces, while a configuration tool accesses the storage hierarchy by the configuration interface. These interfaces oftentimes share common algorithms and methods of storage hierarchy access that are identified as the "Common Components" of the storage hierarchy. The bottom layer of the logical layout of interfaces depicts the hardware abstraction layer interacting with the storage medium by way of middleware. This middleware comprises the device drivers and specifically written programs that translate HAL commands into actionable storage medium read/write instructions. These interfaces are described in detail after a description of the storage hierarchy's internal structures.

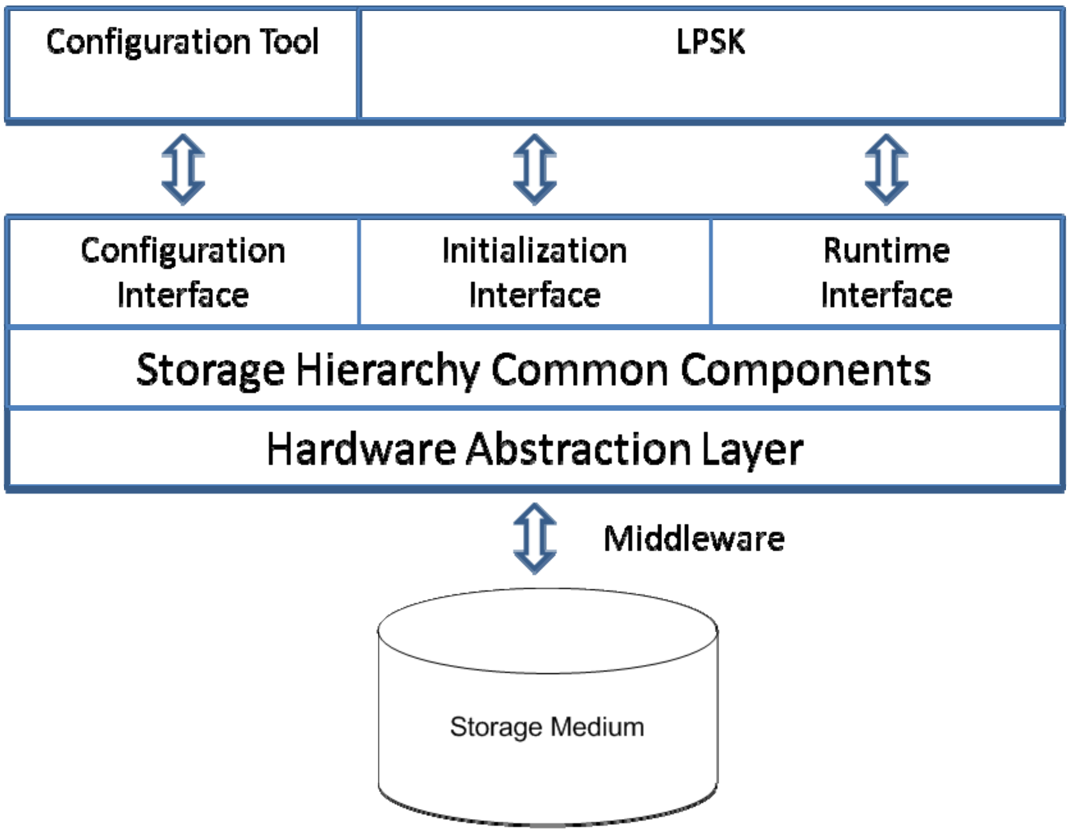


Figure 4. Storage hierarchy logical interface layout.

1. Internal Structure

The storage hierarchy organizes data segments in two ways. The first is by the physical location of data segments across the storage medium on which the storage hierarchy resides. The second is by data segments' logical locations within the tree structure of the storage hierarchy. Both kinds of organization are achieved with data structures that are stored together with the data segments on the storage medium.

LPSK systems may be periodically taken offline for maintenance and configuration. As the resource needs of subjects change, system operators may wish to modify an LPSK system's storage hierarchy by creating or removing data segments during these periods of maintenance. Section 12 of the LPSK functional specification, entitled "LPSK Offline Manager," reinforces this notion of an evolving storage hierarchy configuration: "The LPSK Offline Manager shall provide a user interface to manage existing secondary storage segments, including the display, deletion, initialization, duplication, movement, backup, and restoration of segments [14].” The LPSK Offline Manager is a configuration tool that uses the storage hierarchy configuration interfaces to modify a storage hierarchy.

a. *Physical Organization & Clustering*

The concept of partitioning a storage medium’s space into a number of blocks, or clusters of disk sectors in the case of hard disk drives, was introduced in Chapter II. The storage hierarchy interacts with an abstract view of its storage medium by reading and writing clusters. These clusters are the logical file system “blocks” described by Golden and Pechura, and are comprised of eight disk sectors of 512 bytes each for a total of 4 kilobytes per cluster. In Chapter II, an example file was shown to be allocated across noncontiguous blocks of a file system. Similarly, data segments are physically allocated across contiguous or noncontiguous clusters of a storage medium. The constituent clusters of a data segment are recorded in a data structure that organizes the physical location of all data segments of a storage hierarchy.

The physical location of a cluster of the storage medium refers to its location in the *logical* span of sectors presented by the storage medium. This span of sectors and the collection of sector clusters are both ordered sets upon which an index is imposed. Figure 5 illustrates this concept using two-sector sized clusters for the sake of brevity. In this example, the storage hierarchy begins grouping sectors into clusters beginning at an arbitrary offset of four sectors from the beginning of the storage medium.

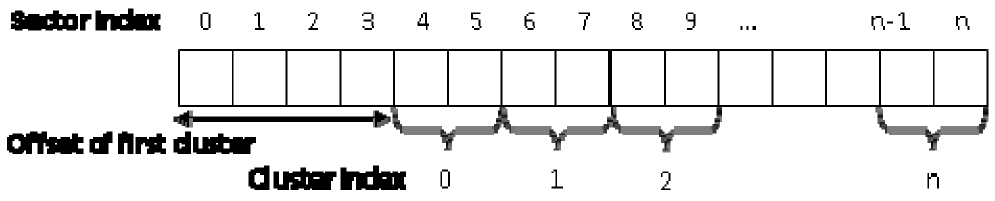


Figure 5. Logical view of storage medium and two-sector cluster assignment

A cluster's location is found by adding the cluster offset to the product of the cluster's index and the number of sectors per cluster:

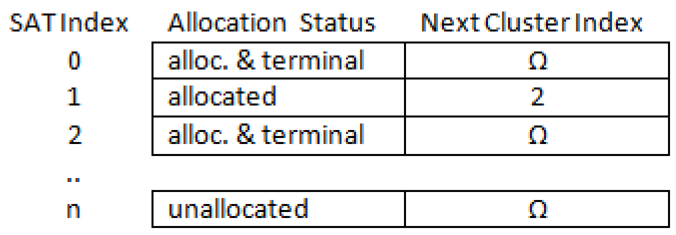
$$cluster\ location = offset\ of\ first\ cluster + (cluster\ index\ \times sectors\ per\ cluster)$$

Using this equation, the third cluster can be found beginning at sector index eight.

b. Segment Allocation Table

The segment allocation table (SAT) is the storage hierarchy internal data structure responsible for the physical organization of data segments on the storage medium. It is a table that records the allocation status of every cluster of the storage medium assigned to the storage hierarchy. The allocation status of a given cluster represents one of three states: unallocated, allocated, or allocated and terminal. *Unallocated* clusters are available for allocation to data segments created during configuration, while *allocated* clusters are used to store all or part of a data segment. If a given cluster is allocated but not terminal, there is an additional cluster allocated to the same data segment. The SAT stores the index value for this next cluster in the "Next Cluster Index" field. Looking up the SAT row that corresponds to this next cluster index reveals its allocation status and potentially the SAT index of yet another cluster. In this way a list or chain of clusters can be read from the SAT by recursive lookup based on the index value for the first cluster allocated to a data segment. A cluster that has a status of *allocated and terminal* is the last cluster of the chain.

Figure 6 illustrates an example SAT where three clusters have been allocated to two data segments. The first cluster at index zero is allocated to a data segment and is that data segment's only cluster. The cluster at index one is marked as allocated but not terminal. The "Next Cluster Index" field shows that the next cluster in the chain can be found at SAT index two. Looking up the cluster at SAT index two reveals that this cluster is last in a two-cluster chain.



SAT Index	Allocation Status	Next Cluster Index
0	alloc. & terminal	Ω
1	allocated	2
2	alloc. & terminal	Ω
..		
n	unallocated	Ω

Figure 6. Example segment allocation table. Omega denotes empty SAT index value.

A SAT entry is comprised of a 2-bit field that contains the allocation status of a cluster and a 30-bit field that contains the SAT index of the next cluster. Table 1 contains the hexadecimal values for the allocation status field.

Status	Value
Unallocated	0x0
Allocated	0x1
Allocated & Terminal	0x2

Table 1. Allocation Status field values.

The SAT data structure is located on the storage medium in reserved space, before space designated for storing data segments. During LPSK system initialization, cluster chains belonging to data segments which have been made known are parsed from the SAT and loaded into memory. During configuration, the SAT is modified to reflect the creation or deletion of data segments. Both of these activities require a priori knowledge of the SAT index for the first cluster of every data segment. This information is stored in a metadata structure that describes the logical layout of data segments in the tree-like structure of the storage hierarchy, which is described next.

c. Directory Table

Directory table data structures are used to reference and organize data segments in terms of the logical hierarchy of the storage hierarchy. A directory table exists for each parent node of the tree and is used to record information about child nodes, including each child node's first cluster SAT index, actual size in bytes, and other metadata. If a parent node has ten child nodes, its directory table will contain ten rows. Table 2 describes the type and size of the fields of the directory table structure.

Directory Table		
Byte Offset	**Size (in bytes)**	**Description**
0x00	4	Name of node
0x04	4	First SAT index
0x08	4	Actual size (in bytes)
0x0C	32	256-bit hash (MAC)
0x2C	1	Type flag (parent or data segment)

Table 2. Directory table structure description.

The name of a node is an unsigned 32-bit integer from zero up to the preconfigured maximum specified during configuration of the LPSK system. The first SAT index field contains the index of the row of the SAT that refers to a data segment's first cluster. With this information gathered from a directory table entry, SAT cluster chains can be parsed and data segments located on the storage medium. The size of data segments may not necessarily be a multiple of the storage hierarchy cluster size. Therefore the actual data segment size as specified during configuration is stored to inform storage hierarchy interfaces when to ignore the unused portions of clusters. A 256-bit field is used to store the MAC generated to ensure data segment integrity.

The last field of the directory table brings about an important result: Directory table structures are stored in the storage hierarchy in the same way as data segments. The type flag field determines if a child node is treated as a data segment or a directory table node by the storage hierarchy. If the type flag is set to "data segment," the node referenced is a leaf node containing a data segment. If the type flag is set to "parent," the child node contains a directory table structure. The unsigned integer value 0x1 denotes a data segment node while the value 0x2 denotes a directory table node. Directory tables are special internal storage hierarchy segments that are never revealed externally. Directory tables may consume more than one cluster in the same manner that data segments do. This occurs if the configured maximum number of child nodes per directory node exceeds the number of directory table rows that can be stored on one cluster.

Each cluster of a directory table contains an additional structure used to store directory table row statistics and the information necessary to follow directory table cluster chains. Table 3 describes the purpose and size of this data. A flag indicates if the directory table is continued in an additional cluster. If it is, the next cluster's SAT index is stored. The total number of table rows a directory table cluster contains is also stored, along with the current number of unused rows in that cluster. A summation of unused rows over all clusters of a directory table is updated and retrieved from the first cluster of the overall directory table cluster chain. By maintaining this information storage hierarchy modules can be prevented from performing futile searches for free directory table rows when there are no unused rows available.

Directory Table Cluster Information		
Byte Offset	**Size (in bytes)**	**Description**
0x00	4	Total number of directory table rows
0x04	4	Number of free, unused rows in this cluster
0x08	4	Overall number of unused rows in all clusters used for directory table
0x0C	4	SAT index of next directory table cluster
0x10	1	More clusters flag

Table 3. Directory table cluster information structure description

The root of the storage hierarchy tree is represented by a directory table that always begins at cluster index zero. This convention provides a static entry point into the storage hierarchy from which all other directory tables and data segments can be located. Figure 7 demonstrates a simple storage hierarchy layout that shows the '/' root node's two children, one of which is a data segment and another which is a directory table node. The directory table node "/5" has two children itself, both data segments.

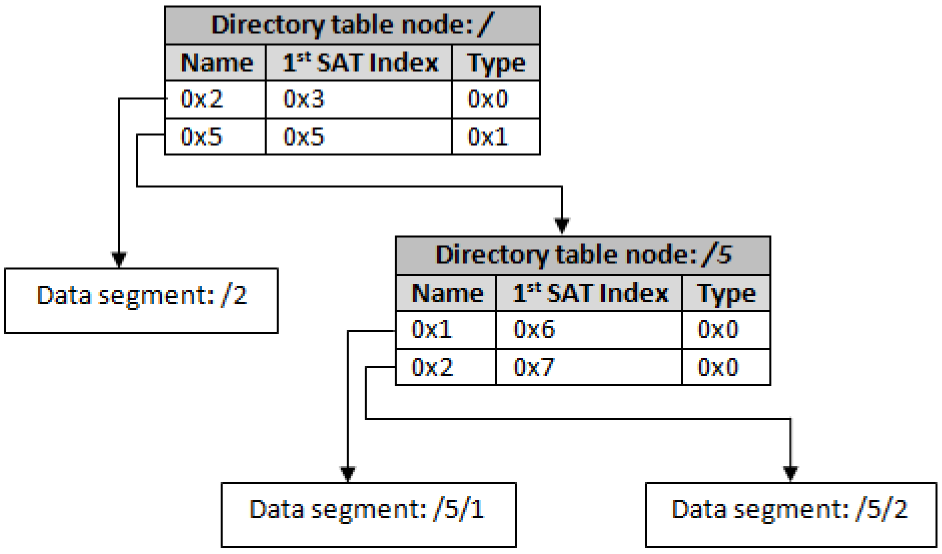


Figure 7. Example logical storage hierarchy layout with directory tables

Directory tables are created and modified during storage hierarchy configuration. They are written to clusters on the storage medium at this time, and may subsequently be rewritten as new child nodes are added to particular parent nodes. During LPSK system initialization, data segments are located by following the paths created by the linking of directory tables, as shown in Figure 6.

d. Other Storage Hierarchy Databases

A storage hierarchy contains several other data structures used internally to maintain state and configuration information. This data is stored on the storage medium in data blocks located before the SAT. Table 4 shows the first of these data structures, which contains information about the layout of the storage hierarchy on the storage medium. The first three fields contain the location of other storage hierarchy databases in terms of their offset into the logical span of the storage medium. The storage hierarchy cluster size is also defined along with the size of the SAT and the maximum number of child nodes per directory node allowed. This data structure is written when an LPSK system's storage medium is first formatted in preparation for configuration.

Static Configuration Data		
Byte Offset	**Size (in bytes)**	**Description**
0x00	4	Offset from start of storage medium to SAT
0x04	4	Offset from start of storage medium to first cluster
0x08	4	Offset from start of storage medium to Cluster database
0x0C	4	Cluster size in number of 512-byte blocks
0x10	4	The size of the SAT in number of 512-byte blocks
0x14	4	The number of SAT rows
0x18	4	The maximum number of child nodes per directory node

Table 4. Static Configuration data structure description

An additional data structure maintains a count of the state of storage hierarchy clusters, including how many clusters are allocated and unallocated. By keeping a count of the total number of free clusters available for allocation, storage hierarchy interfaces do not have to scan the entirety of the SAT to determine if there are enough free clusters every time a new data segment is created. A separation of allocated cluster counts based on what type of data clusters contain—data segments or directory tables—provides some insight into the amount of metadata overhead present in the storage hierarchy, but this information is not required or used by any interfaces. Table 5 enumerates this cluster accounting data structure.

Cluster Values		
Byte Offset	**Size (in bytes)**	**Description**
0x00	4	Total number of clusters
0x04	4	Free/Unallocated clusters
0x08	4	Clusters allocated to data segments
0x0C	4	Clusters allocated to directory tables

Table 5. Cluster values data structure description.

e. Handle Table

The handle table data structure is distinct from all previously discussed data structures in that it only exists in the primary memory of an LPSK system. The handle table is constructed during LPSK initialization and associates handles, or data segment selectors, to particular data segments of the storage hierarchy. Data from the handle table is used during runtime, after initialization, to perform operations on secondary storage data segments. Table 6 displays the contents of this data structure.

Handle Table		
Byte Offset	**Size (in bytes)**	**Description**
0x00	4	Handle
0x04	4	Size (in bytes)
0x08	4	First SAT index
0x0C	4	Parent’s first SAT index
0x10	4	Directory table index

Table 6. Handle table data structure description.

The relevant information of every secondary storage data segment to be made known in an LPSK system, is recorded in the handle table data structure. This information includes the handle by which the LPSK refers to a particular data segment, the data segment’s size in bytes, and the SAT index of its first cluster. This information facilitates the reading and writing of data segments from the storage hierarchy. The SAT index of the first cluster of a data segment’s parent is also recorded, along with the index within the directory table that refers to that data segment. This information allows storage hierarchy interfaces to read and update metadata, such as a data segment’s MAC, during runtime.

The next section describes the interfaces the LPSK and configuration tools use to access the internal data structures described in this section.

2. External Interfaces

As a module of the overall LPSK, the storage hierarchy makes its services available through a number of external interfaces. These interfaces are said to be external because they are designed to be used by other modules of the LPSK. The availability of these external interfaces depends on the three distinct modes of an LPSK system: configuration, initialization, and run-time. In configuration mode, the storage medium on which the storage hierarchy resides is attached to and modified by a non-LPSK system running configuration software. During initialization mode, the storage hierarchy is prepared for operation by an interface that constructs a handle table data structure in primary memory which records all data segments exported to system subjects and data segments used internally by the LPSK. Data segments may also be swapped in during this mode. In the run-time mode, the storage hierarchy exposes interfaces used to flush, swap in, and swap out data segments from secondary storage.

The technical specification of the external interfaces (see Appendix A) categorizes interfaces based on the mode of system operation for which they are meant to be used.

a. Configuration Interfaces

The configuration interfaces of the storage hierarchy are used to construct the layout of a storage hierarchy on a storage medium and populate it with data segments. The collection of configuration interfaces provides the application programming interface (API) that is used by storage hierarchy configuration tools. The thirteen individual interfaces are listed in categories that reflect their usage.

Storage hierarchy layout interfaces:

- `initialize_databases`: Loads storage hierarchy configuration, cluster, and SAT databases into primary memory from the storage medium.
- `make_dseg`: Creates a data segment node off of a parent node.
- `del_dseg`: Removes a data segment node and zeros out its associated clusters.
- `make_subtree`: Creates a directory table node off of a parent node.
- `del_subtree`: Removes a parent directory node and all of its child nodes of all types. This interface descends a sub-tree beginning at the specified directory table node, removing all descendent nodes.
- `get_child`: Returns the names and node types of the specified parent's child nodes. A configuration tool can effectively enumerate a storage hierarchy's layout using this interface.

Data segment access interfaces:

- `open_dseg`: Returns a handle associated with a data segment that is used by other interfaces to access the contents of data segments.
- `write_dseg`: Writes data into the data segment associated with the supplied handle.
- `read_dseg`: Reads data from a data segment associated with the supplied handle.
- `close_dseg`: Generates a new MAC for the data segment associated with the handle then purges that handle from the handle table.

Data segment metadata interfaces:

- `get_dseg_size`: Returns the actual size of a data segment. This interface may be used by a configuration tool to learn how large a buffer to allocate in primary memory prior to reading the contents of a data segment.
- `check_hash`: Compares an integrity hash of the contents of a data segment with the MAC stored in that data segment's metadata.

b. Initialization Interfaces

The initialization interfaces of the storage hierarchy are used to construct and populate the handle table data structure prior to LPSK runtime. Interfaces used to accomplish the *swap in* operation are also provided.

- `initialize_databases`: Loads storage hierarchy configuration, cluster, and SAT databases into primary memory from the storage medium. Initializes an empty handle table in memory with as many rows as necessary to record the specified number of handle to data segment associations.
- `get_handle`: Loads the specified data segment's meta-data into the handle table database and returns a handle to be used for future operations on that data segment.
- `read_in`: Loads the data segment associated with the specified handle from the storage hierarchy storage medium into the specified buffer in primary memory. This interface supports the *swap in* operation.

- `get_dseg_size`: Returns the size of the data segment associated with the specified handle.
- `get_dseg_hash`: Returns the MAC value stored in the meta-data of the data segment associated with the specified handle.

c. *Runtime Interfaces*

The storage hierarchy runtime interfaces are used by the LPSK to perform *swap in*, *swap out*, and *flush* actions on data segments. Both *swap out* and *flush* operations are accomplished using the same interface as they have the same effect from the perspective of the storage hierarchy. With the exception of the `write_out` interface, all runtime interfaces behave the same as their initialization interface counterparts.

- `read_in`: Loads the data segment associated with the specified handle from the storage heiarchy storage medium into the specified buffer in primary memory.
- `write_out`: Writes the data segment from primary memory associated with the specified handle from the specified buffer, then calculates and stores the new data segment MAC. This interface is used by the *swap out* and *flush* operations.
- `get_dseg_size`: Returns the size of the data segment associated with the specified handle.
- `get_dseg_hash`: Returns the MAC value stored in the meta-data of the data segment associated with the specified handle.

3. Hardware Abstraction Layer

The storage hierarchy HAL is designed to provide an abstract and simple view of the storage medium. It provides the interfaces to the storage medium used to read and write clusters, SAT data, and storage hierarchy configuration databases.

- `read_cluster`: Reads the specified cluster into memory.
- `write_cluster`: Writes data to the specified cluster.
- `read_dir`: Reads the specified cluster containing a directory table structure into memory.
- `write_dir`: Write a directory table structure to the specified cluster.

- `read_blocksize`: Read a disk block-sized data structure into memory.
- `write_blocksize`: Write a disk block-sized data structure to disk

C. IMPLEMENTATION METHODOLOGY

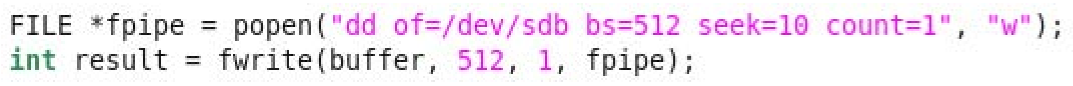
The storage hierarchy external interfaces and common components were developed in ANSI-C89 standard programming code with the assistance of the Apache Software Foundation licensed Eclipse Platform. The Eclipse Platform is an integrated development environment that provides various features to assist developers in the rapid development and compilation of code. An Eclipse project is the collection of program source files and additional internal databases that contain meta-data about various aspects of the program. Eclipse recognizes and records program function and variable names and structures in its databases as they are authored to provide helpful functionality to programmers. Such functionality includes auto-completion of C structure members and generation of function call hierarchies. The development and testing platform consisted of a Fedora Linux (kernel version 2.6.31.5) workstation. The GNU Compiler Collection (GCC) C compiler was used to produce binary executable modules of storage hierarchy configuration interfaces and common components for the purposes of testing their functionality and correctness.

An Eclipse C project was created as a collection of C program source and header files that together incorporate the storage hierarchy external interfaces and common components. C pre-processor conditional directives embedded within each module determine if module functions are compiled as part of a configuration tool or the LPSK binary. The distinction between LPSK and non-LPSK context modules and functions is described later in this section.

1. Storage Medium Construct

The storage hierarchy prototype requirements call for a test construct located in primary memory or secondary storage of a test machine that mimics the storage hierarchy's storage medium. This simulated storage medium is necessary during development and testing of the prototype to ensure the common components used by external interfaces correctly read and modify hierarchy structures. It is also possible to test for storage hierarchy structure and data segment permanence and retrievability if the construct is located on the test machine's secondary storage.

The special construct was created using a virtual hard disk drive which is attached to the test machine as a raw, unformatted disk device. The Unix system command line tool *dd* is used to read and write from the virtual disk. The dd tool allows reading and writing raw binary data from block I/O devices without regard for file systems or partitioning conventions. The storage hierarchy prototype hardware abstraction layer implementation invokes the *dd* tool by constructing a command line string and calling the C library function `popen()`. This function executes the supplied string as a Linux shell command and returns a file stream from the process. The arguments supplied to *dd* through the command line string tell *dd* where to read and write from the disk device. Figure 8 depicts the sequence of C instructions that accomplishes this.



```
FILE *fpipe = popen("dd of=/dev/sdb bs=512 seek=10 count=1", "w");
int result = fwrite(buffer, 512, 1, fpipe);
```

Figure 8. Example HAL *dd* string that writes a block of 512 bytes from the pointer `buffer` to the device `/dev/sdb`

D. IMPLEMENTATION

The storage hierarchy interface prototypes developed in this thesis are compiled and linked based on which external software program uses them. When interfaces are used by the LPSK binary, those interfaces are said to be executed in an LPSK context. The configuration interfaces execute in a non-LPSK context during the configuration of an LPSK system and contain the program logic necessary to perform all storage hierarchy maintenance operations. The initialization and runtime interfaces execute in the LPSK context and perform a subset of these operations. Both storage hierarchy interface contexts share common components such as helper functions and data structures. These functions and data structures are used by the interfaces to perform common operations, such as navigating the storage hierarchy tree structure, or retrieving data segment meta-data. The module dependencies of both interface contexts are introduced next, followed by a description of the common components.

1. Configuration Interface Implementation

Figure 9 illustrates the module dependencies of the storage hierarchy configuration interfaces and common components. Each diagram element represents a C program module. The common components consist of the modules used by the configuration interfaces. Modules are layered in terms of their dependencies. For example, several configuration interfaces depend on the handle table manager module to provide data segment handles. The handle table manager in turn depends on the directory manager and SAT manager modules to retrieve directory table structures and meta-data associated with the desired data segment. These two modules depend on the HAL module, which interfaces with the storage hierarchy's storage medium.

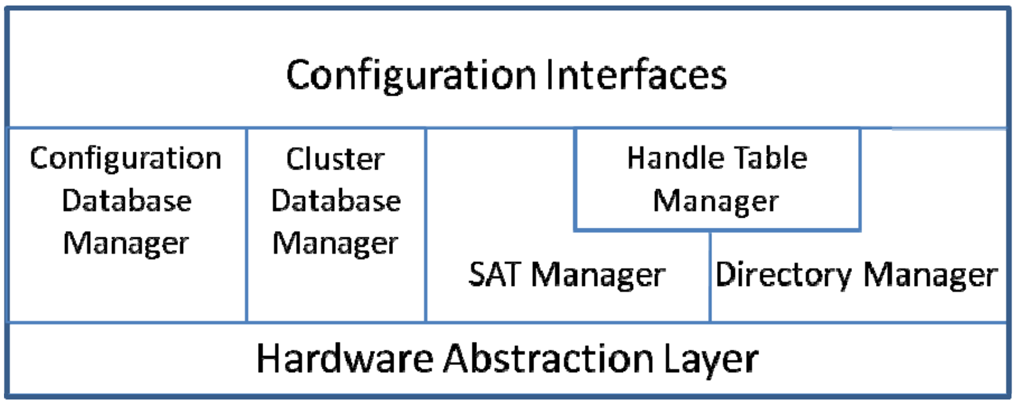


Figure 9. Configuration interface module dependency diagram.

2. Initialization and Runtime Interface Implementation

Figure 10 illustrates the module dependencies of the storage hierarchy initialization and configuration interfaces and common components. In contrast to the configuration interface dependency diagram, the initialization and runtime interfaces no longer have direct access to the SAT and directory manager modules, and may only interact with the limited facilities provided by the handle table manager. The cluster database manager is excluded entirely, as the statistics it maintains are useful only in the creation and deletion of data segments and directory structures. This reduction in modules reflects the reduction in the scope of storage hierarchy operations performed by these interfaces.

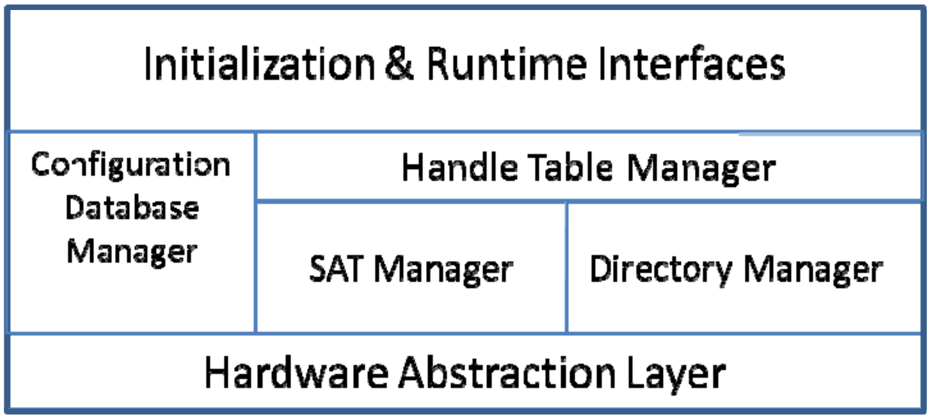

Initialization & Runtime Interfaces
Configuration Database Manager
Handle Table Manager
SAT Manager
Directory Manager
Hardware Abstraction Layer

Figure 10. Initialization and Runtime interface module dependency diagram including.

3. Common Components

The storage hierarchy common components consist of six individual C program modules that maintain data and perform actions necessary for the operation of various interfaces. Each module function is identified by which context it is used. Each module is described in detail in the following sections.

a. Configuration Database Manager

This module is responsible for maintaining the configuration database in memory, providing functions to retrieve configuration data that describe the layout of data structures on the storage medium, and the dimensions of certain constructs.

`Config_db.c` external functions		
Name	**Description**	**Context**
`init_config_db`	Loads the configuration database into memory from disk.	Both
`get_SAT_offset`	Returns the block offset value to the beginning of the Segment Allocation Table database on disk.	Both
`get_cluster_offset`	Returns the block offset value to the beginning of the storage medium where data segments and directory table clusters are located.	Both
`get_cluster_size`	Returns the size of clusters in number of 512-byte disk blocks.	Both
`get_cluster_db_offset`	Returns the block offset value to the cluster database structure on disk.	Both
`get_SAT_size`	Returns the size of the Segment Allocation Table in terms of 512-byte disk blocks.	Both
`get_SAT_records`	Returns the number of 32-bit rows in the SAT	Both
`get_MAX_children`	Returns the maximum number of children per directory node as defined by an LPSK configuration vector	Both

Table 7. Configuration database manager external functions description.

b. Cluster Database Manager

This module is responsible for maintaining the cluster database in memory and providing functions to retrieve and update cluster values. Cluster usage values are used by configuration interfaces only.

`cluster_db.c` external functions		
Name	**Description**	**Context**
`init_cluster_db`	Loads the cluster database into memory from disk.	Non-LPSK
`get_free_clusters`	Returns the number of free clusters available in the storage hierarchy.	Non-LPSK
`inc_dseg_clusters`	Increases the data segment cluster value by the specified amount and decreases the free cluster value by the same amount, writes the cluster database to disk.	Non-LPSK
`dec_dseg_clusters`	Decreases the data segment cluster value by the specified amount and increases the free cluster value by the same amount, writes the cluster database to disk.	Non-LPSK
`inc_dir_clusters`	Increases the directory table cluster value by the specified amount and decreases the free cluster value by the same amount, writes the cluster database to disk.	Non-LPSK
`dec_dir_clusters`	Decreases the directory cluster value by the specified amount and increases the free cluster value by the same amount, writes the cluster database to disk.	Non-LPSK

Table 8. Cluster database manager external functions description.

c. Directory Manager

This module provides functions related to the storage and access of directory table structures and node records. Functions are provided that allow iteration along a path in the storage hierarchy, search for the records of particular data segment and directory nodes, and retrieval of data segment metadata.

`directory_mgr.c` external functions		
Name	**Description**	**Context**
`dir_retrieve_dir`	Populates the supplied directory table structure in memory from a directory table cluster at the specified SAT index from disk.	Both
`dir_write_dir`	Writes a directory table structure to the cluster at the specified SAT index on disk.	Both
`dir_retrieve_record`	Returns a directory table record associated with the specified node from the supplied directory table structure in memory to the caller.	Both
`dir_retrieve_record_and_index`	Returns a directory table record associated with the specified node from the supplied directory table structure in memory. It also returns the index in the directory table where the record was found.	Both
`dir_check_node_Presence`	Returns `TRUE` if the specified node is found in the directory table cluster chain headed by the directory table referenced by supplied pointer.	Both
`dir_get_hash`	Returns the hash value of the specified node from its directory table record.	Both

`directory_mgr.c` external functions		
Name	**Description**	**Context**
`dir_get_size`	Returns the size of the specified data segment node	Both
`dir_rtr_dir_by_path`	Returns the node type and SAT index of the last node of the supplied path after it has walked through the storage hierarchy tree according to the supplied path string.	Both
`dir_update_hash`	Updates the 256-bit hash of the data segment located in the specified row of the supplied directory table with the supplied hash.	Both
`dir_get_type`	Returns the node type of the specified node.	Non-LPSK
`dir_new_directory`	Creates a new directory table structure in memory.	Non-LPSK
`dir_delete_record`	Removes the record associated with the specified node from the supplied directory table. Returns the SAT index of the node's first cluster	Non-LPSK
`dir_get_nth_record`	Returns the node name and type of the n'th occupied row of the supplied directory table.	Non-LPSK
`dir_rtr_node_SAT_ index`	Returns the SAT index of the specified node's first cluster from the supplied directory table.	Non-LPSK
`dir_ins_record_and_ commit`	Inserts a new node record into a directory table structure and writes associated directory table clusters to disk.	Non-LPSK
`dir_get_num_free_ Records`	Returns the number of free records (and therefore child nodes that can be created) for a directory table structure cluster chain headed by the directory table referenced by supplied pointer	Non-LPSK

Table 9. Directory manager external functions description

d. Handle Manager

This module is responsible for maintaining the handle table manager in memory. The handle table is used in both code bases to associate a data segment with a handle that is then used for read/write operations.

`Handle_mgr.c` external functions		
Name	**Description**	**Context**
`init_handle_manager`	Constructs a dynamically sized handle-to-data segment table in memory.	Both
`get_handle`	Loads a data segment's pertinent meta-data into the handle table and returns a handle used to reference that data segment.	Both
`delete_handle`	Purges a handle table row from the handle table.	Both
`write_to_handle`	Retrieves meta-data from the handle table row associated with the specified handle and calls SAT manager module to write to the storage medium.	Both
`read_from_handle`	Retrieves meta-data from the handle table row associated with the specified handle and calls SAT manager module to read from storage medim.	Both
`get_handle_size`	Returns the size of the data segment associated with the specified handle.	LPSK
`get_handle_hash`	Returns the 256-bit hash value of the data segment associated with the specified handle	LPSK
`calc_hash`	Calculates and returns the hash value of the contents of the data segment associated with the specified handle.	Both
`update_hash`	Updates the 256-bit hash of the data segment associated with the specified handle.	Both

Table 10. Handle table manager external functions description.

e. SAT Manager

This module is responsible for maintaining the Segment Allocation Table in memory and providing functions to allocate and unallocate cluster chains, and read and write the SAT to the storage medium. It also provides functions to read and write clusters and data segments from the storage medium.

`SAT_mgr.c` **external functions**		
Name	**Description**	**Context**
`sat_init`	Loads the SAT into memory.	Both
`sat_alloc_clusters`	Creates a cluster chain of the specified size by setting allocation status and next SAT index values. Commits SAT changes back to disk. Returns the SAT index of the head of the chain.	Non-LPSK
`sat_unalloc_clusters`	Destroys a cluster chain starting at the specified SAT index by resetting the allocation status for each member of the chain. Commits SAT changes back to disk. Returns the number of SAT rows	Non-LPSK
`sat_get_next`	Returns the next SAT index value of the specified SAT index.	Both
`sat_read_cluster`	Reads the cluster with the specified SAT index into memory.	Both
`sat_write_dseg`	Writes data from the specified buffer to the data segment located with the specified handle on the storage medium.	Both
`sat_read_dseg`	Reads data from the data segment associated with the specified handle from the storage medium into the specified buffer.	Both

Table 11. SAT manager external functions description.

E. SUMMARY

This chapter introduced the requirements of the storage hierarchy prototype and presented a design. The implementation methodology was discussed and the external interfaces and common component external functions were enumerated. The next chapter contains test procedures and results for the external interfaces.

IV. TESTING

This chapter presents an LPSK storage hierarchy scenario that is used to test the storage hierarchy prototype external interfaces. The results of each external interface test are grouped based on their interface category used throughout this work.

A. TEST SCENARIO

The test scenario consists of a 128-megabyte virtual disk that is pre-formatted to contain the prerequisite storage hierarchy configuration databases and segment allocation table. This relatively small virtual disk size (compared to modern disk drives) was chosen to make test cases involving disk space limitations easier to construct. The disk is formatted by a program that utilizes its knowledge of storage hierarchy structures and the storage hierarchy HAL interface to write these databases, SAT, and directory tables to disk before configuration interfaces are used. The specific data used to create the formatted test disk are described next, followed by a brief description of the programs that simulate a storage hierarchy configuration and LPSK interface user.

1. Storage Medium Format

Table 12 contains the configuration database values for the test storage hierarchy. The logical disk locations of various storage hierarchy data structures for the test scenario were chosen arbitrarily as storage hierarchy requirements do not describe the organization of data segments or meta-data on the storage medium.

Scenario Configuration Data		
Name	**Value**	**Description**
SAT_offset	10	Offset from start of storage medium to SAT
first_cluster_offset	266	Offset from start of storage medium to first cluster
cluster_db_offset	9	Offset from start of storage medium to Cluster database
cluster_size	8	Cluster size in number of 512-byte blocks
SAT_size	256	The size of the SAT in number of 512-byte blocks
SAT_records	32,768	The number of SAT rows
max_children	128	The maximum number of child nodes per directory node

Table 12. Storage hierarchy test scenario configuration database data.

The SAT begins at the tenth block of the virtual disk and spans 256 disk blocks. It contains a total of 32,768 rows. This SAT size is large enough to address a 128-megabyte storage medium space. The maximum number of child nodes is set to 128 nodes.

Table 13 contains the cluster database values for the test storage hierarchy.

Scenario Cluster Values		
Name	**Value**	**Description**
total_clusters	32,768	Total number of clusters
free_clusters	32,766	Free/Unallocated clusters
dseg_clusters	0	Clusters allocated to data segments
dir_clusters	2	Clusters allocated to directory tables

Table 13. Storage hierarchy test scenario cluster database values.

The value for the total number of clusters matches the number of SAT rows from the configuration database, since each SAT row is mapped to a cluster. The free cluster value is two less, as two clusters are pre-allocated to the root directory node, which is always present in a storage hierarchy. The directory cluster value reflects the allocation of two directory clusters.

Recall from Chapter III the directory table and directory table cluster information structures. The directory table cluster information structure, which is located on every cluster allocated for storage of directory tables, consumes 20 bytes. One row of a directory table structure consumes 48 bytes. The maximum number of directory table rows per directory table cluster is found by subtracting the size of the directory table cluster information structure from the size of clusters in bytes, divided by the size of a directory table row:

$$max\ directory\ rows = \frac{cluster\ size - directory\ cluster\ information\ size}{directory\ row\ size}$$

Given a four kilobyte cluster size, the maximum number of directory table rows per directory table cluster is $84.91\overline{6}$. The remainder is ignored to yield 84 directory table rows per directory table cluster. Two directory table clusters are required to store the meta-data in order to accommodate the maximum number of child nodes, 128, permitted by the configuration.

The root directory table is constructed from two directory table structures. The first directory table's cluster total number of records value is set to 84. The more clusters flag is set to indicate that another directory table cluster follows. The location of the next directory table cluster is set to SAT index one. The maximum records value is set to 128 to indicate how many directory table rows are available for use across all directory table clusters of the chain. Each directory table row is initialized to null values and their statuses are set to `NODE_TYPE_FREE` to indicate they are available for usage. The directory structure is then written to disk using a HAL interface. The second directory table cluster is constructed and written to disk at the next cluster in the same way. The cluster directory `more` flag is unset to indicate no more directory clusters follow.

The segment allocation table is constructed as an array of 32-bit unsigned integers of length equal to the number of SAT rows value stored in the configuration database, plus one. The extra integer contains a special value that is used by the SAT_init function of the SAT manager module to ensure the binary data is has retrieved from disk during initialization represents a SAT. The SAT array is written to disk using a HAL interface.

2. Test Programs

Two test programs simulate both a storage hierarchy configuration tool and the LPSK. The configuration interface test program uses configuration interfaces to create and modify the storage hierarchy structure on disk, read and write data segments, compare data segment hash values, etc. The LPSK test program uses initialization and runtime interfaces to get handles for data segments created during configuration and perform a sequence of read and write operations, comparing expected hash values with those stored in meta-data. The test programs record the return value for each interface call and any unexpected results it encountered when reading and writing data segments. In addition, both test programs make use of the `format` function provided by the program that returns the test storage hierarchy's storage medium to its original state.

Each test program is produced by invoking the storage hierarchy *make file* with a specific argument. A *make file* contains the compiler and linker directives and is interpreted by the *Make* utility to invoke the tools necessary to produce machine-code binary files. *Make* is part of the GNU Compiler Collection [17]. The `config_test` argument instructs the *Make* program to compile and link the configuration test program with the storage hierarchy configuration interfaces. The configuration test program performs each configuration group autonomously and can also be directed to perform some of the storage hierarchy configuration tasks required by the LPSK test program. Figure 11 illustrates the hierarchical dependencies of the configuration test program

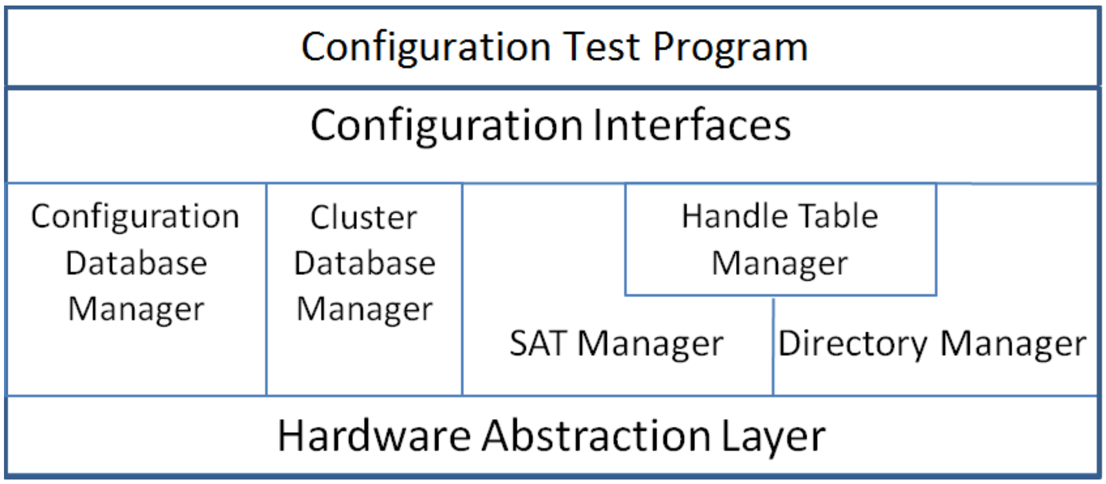


Figure 11. Configuration test program compiled with configuration interfaces and common components as a single monolithic executable.

The LPSK test program is produced by invoking the *make file* with the `lpsk_test` argument. The LPSK test program is linked with the storage hierarchy initialization and runtime interfaces, and performs the tests involving these interfaces. Figure 12 illustrates the hierarchical dependencies of the LPSK test program.

The storage hierarchy interface calls that are performed by the test programs are reflected in each test group's test procedure, found in Appendix B. Both test programs expect an argument with a value that corresponds to a specific test group, or a set of steps that make up part of a test group. When a test program is executed with a valid test instruction, it performs the test procedures associated with the specified test group and prints the results. To run the `sh_c_make_dseg` test procedure, for example, the configuration test program is first generated by invoking the storage hierarchy *make file* with argument `test_config`. The resulting executable file is then called with test group argument `TEST_C2`. Certain initialization and runtime interface test groups require that both test programs be called in sequence as described by the test procedure.

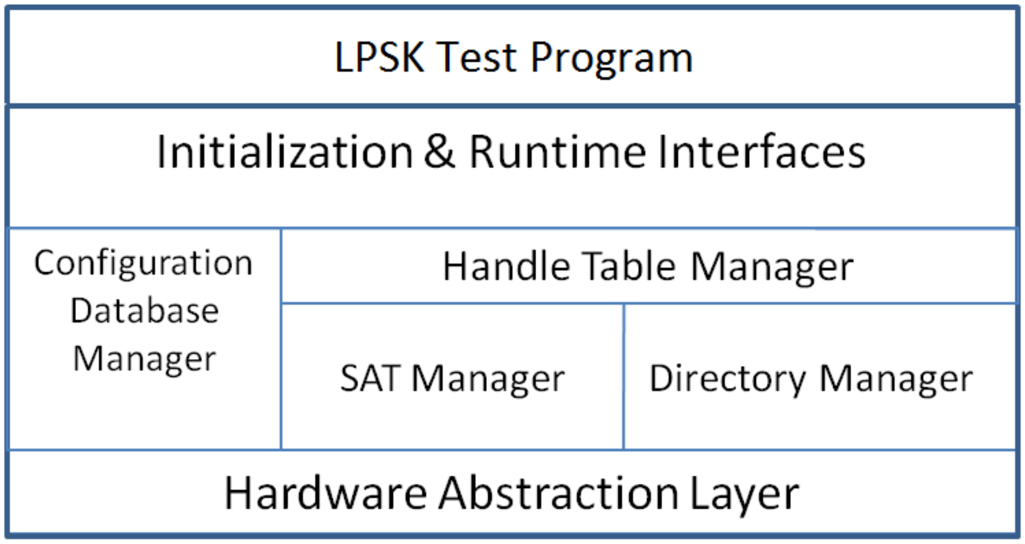

LPSK Test Program
Initialization & Runtime Interfaces
Configuration Database Manager
Handle Table Manager
SAT Manager
Directory Manager
Hardware Abstraction Layer

Figure 12. LPSK test program compiled with initialization and runtime interfaces and common components as a single monolithic executable.

B. EXTERNAL INTERFACE TESTING

The storage hierarchy prototype external interfaces were tested through a series of "black-box" tests. Black-box testing relies on the documentation of the external interfaces to confirm whether or not they behave according to specification. Positive and negative tests are performed for each interface. *Positive tests* demonstrate that interfaces function correctly on valid input parameters. *Negative tests* demonstrate that the external interfaces return the expected error codes when invalid input or error states are encountered. The external interfaces are tested on the storage hierarchy-formatted virtual disk described in Section A. See Appendix B for the test plans used to produce the results that follow.

Multiple tests were performed for each interface. Individual tests were labeled with a name and test type and are recorded in a table along with the relevant parameters supplied to the interface, the expected result of the interface, and a record of the observed behavior of the interface. Each test was performed independently of others except where indicated by the test naming scheme. Tests were named by their category (configuration, initialization, runtime), a unique test group number that distinguish different interfaces tests, and a sequence of capital letters that distinguished individual tests. When tests depend on and need to occur after previous tests, the capital letter remains the same and a sequentially increasing digit is affixed. For example, test C1A is the first test of a configuration interface, and test C2A is the first test of the next configuration interface. Whereas, tests labeled C1A.1 and C1A.2 imply that the results of the latter rely on the results of the former.

Tests are categorized by their type, either positive or negative. Only interface parameters relevant to a specific test run are enumerated (See Appendix A for full list of external interface parameters). Parameters that are not meaningful for a specific test were omitted. In the case of positive tests, an interface return value indicating that no error has occurred is expected. Specific error codes are expected as the result of negative tests. Many interfaces that rely on previous storage hierarchy operations, i.e., the `sh_c_write_dseg` interface cannot be tested for functional correctness unless a handle has been associated with a newly created data segment. Test prerequisites are described for these context-sensitive interfaces.

1. Configuration Interface Testing

Table 14 contains the results of test group 1, the `sh_c_initialize_databases` interface. The configuration database is located at block offset eight as described in Section A of this chapter.. Passing any other block offset value will cause the configuration database initialization function to fail. For test runs C1E and C1F, the block offset data stored in the configuration database was set to verify that the module correctly identifies the error.

Table 15 contains the results of test group 2, the `sh_c_make_dseg` interface. These tests were performed after storage hierarchy databases and structures had been initialized. Table 16 contains the results of test group 3, `sh_c_make_subtree` interface. These tests were also performed after storage hierarchy initialization. A large data segment was created to consume all free clusters prior to test run C3C to generate the `SH_C_DISKSPACE` error code.

Test group 4, the `sh_c_open_dseg` interface, requires the presence of data segments in order to perform meaningful functional tests. Table 17 contains the test results for this interface after the storage hierarchy was initialized and three data segments were created at paths “/1”, “/2”, and “/3”. The number of handle table rows in the configuration database was set to two during initialization to cause the `SH_C_HANDLE_TABLE_FULL` error code to be generated when `sh_c_open_dseg` is called for a third data segment.

Test group 1: `sh_c_initialize_databases`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C1A	Pos.	block_offset: 8	No error, database structures loaded.	Pass	Offset 8 is the correct value
C1B	Neg.	block_offset: 0	`SH_C_CONFIG_DB_LOAD_ERR`	Pass	
C1C	Neg.	block_offset: 8	`SH_C_CLUS_DB_LOAD_ERR`	Pass	Configuration database altered to reflect incorrect location of cluster database.
C1D	Neg.	block_offset: 8	`SH_C_SAT_LOAD_ERR`	Pass	Configuration database altered to reflect incorrect location of SAT

Table 14. Test group 1: sh_c_initilizase_databases.

Test group 2: `sh_c_make_dseg`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C2A.1	Pos.	path: "/" node: 0 size: 9000	No error, data segment created	Pass	Data segment node at path "/0" created.
C2A.2	Neg.	path: "/" node: 0	`SH_C_NODE_ALREADY_EXISTS`	Pass	Duplicate data segment node from test C2A.1 detected.
C2B	Neg.	path: "/" node: 128	`SH_C_MAX_NODE_NAME`	Pass	Specified node name exceeds configured maximum of 128 nodes.
C2C	Neg.	path: "/" node: 0 size: 134217728	`SH_C_DISKSPACE`	Pass	Data segment size exceeds available clusters.
C2D	Neg.	path: "a"	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C2E	Neg.	path: "/a"	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C2F	Neg.	path: "/0"	`SH_C_PATH_ERROR`	Pass	Input "/0" is not a valid path.

Table 15. Test group 2: `sh_c_make_dseg`.

Test group 3: `sh_c_make_subtree`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C3A.1	Pos.	path: "/" node: 0	No error, directory table created	Pass	
C3A.2	Neg.	path: "/" node: 0	`SH_C_NODE_ALREADY_EXISTS`	Pass	Duplicate directory table node detected.
C3B	Neg.	path: "/" node: 128	`SH_C_MAX_NODE_NAME`	Pass	Specified node name exceeds configured max of 128 nodes.
C3C	Neg.	path: "/" node: 0	`SH_C_DISKSPACE`	Pass	Available disk space consumed prior to this test.
C3D	Neg.	path: "a"	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C3E	Neg.	path: "/a"	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C3F	Neg.	path: "/0"	`SH_C_PATH_ERROR`	Pass	Input "/0" is not a valid path.

Table 16. Test group 3: sh_c_make_subtree.

Test group 4: `sh_c_open_dseg`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C4A.1	Pos.	path: “/” node: 1	No error, handle zero returned.	Pass	/1 was created prior to this test
C4A.2	Pos.	path: “/” node: 2	No error, handle one returned.	Pass	/2 was created prior to this test
C4A.3	Neg.	path: “/” node: 3	`SH_C_HANDLE_TABLE_FULL`	Pass	All handles were consumed prior to this test.
C4B	Neg.	path: “/” node: 10	`SH_C_NO_SUCH_NODE`	Pass	No node exists at path “/10”
C4C	Neg.	path: “a”	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C4D	Neg.	path: “/a”	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C4E	Neg.	path: “/0”	`SH_C_PATH_ERROR`	Pass	Path does not exist

Table 17. Test group 4: `sh_c_open_dseg`.

Table 18 contains the results for test group 5, the `sh_c_write_dseg` interface. Two data segments of size 9,000 bytes were created and associated with handles. Two buffers, the first containing 9,000 bytes of pre-generated binary data, and the second containing 9,000 bytes of different pre-generated binary data, were created in the memory of the test system. Hashes were generated for each buffer, using the same technique used to generate the MAC for data segments in the storage hierarchy. The `sh_c_write_dseg` interface was then called on the first data segment handle to write the entirety of the first buffer to the data segment. The second data segment handle was written in two halves from the second 9,000 byte buffer. This tested the offset and byte count parameters that allow the interface caller to write the specified number of bytes at the specified byte offset in the data segment.

Table 19 contains the results for test group 6, the `sh_c_read_dseg` interface. Tests *C6A* and *C6B* from this group are performed immediately after the corresponding tests *C5A* and *C5B* in test group 5. The contents of the two data segments that were previously written are read into buffers by the `sh_c_read_dseg` interface. New hashes are generated based on the contents of these buffers and compared to the hash values generated by the `sh_c_write_dseg` test group. By that showing these hash values are equivalent, the `sh_c_read_dseg` interface is shown to correctly read the contents of the correct data segments.

Table 20 contains the results for test group 7, the `sh_c_close_dseg` interface. Test *C7A* uses the data segment opened and written from test *C5A*. When a data segment is closed, the hash value of its contents is generated and stored. The generated hash is compared to the hash value generated in test group 5 and found to be equivalent, demonstrating that `sh_c_close_dseg` correctly generates the hash of the data segment associated with the specified handle.

Table 21 contains the results for test group 8, the `sh_c_delete_dseg` interface. A data segment at path “/1” and a directory table at path “/2” were created for this test. The interface was tested by calling it to remove the data segment.

Test group 5: `sh_c_write_dseg`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C5A	Pos.	handle: 0 buffer: buf_ptr1 byte_count: 9000 offset: 0	No error.	Pass	All 9,000 bytes of buf_ptr1 is written to the data segment
C5B.1	Pos.	handle: 1 buffer: buf_ptr2 byte_count: 4500 offset: 0	No error.	Pass	4,500 bytes from buf_ptr2 is written to the beginning of the data segment
C5B.2	Pos.	handle: 1 buffer: buf_ptr2 byte_count: 4500 offset: 4500	No error.	Pass	4,500 bytes from buf_ptr2 is written to the data segment beginning at byte offset 4,500.
C5C	Neg.	handle: 2	`SH_C_HANDLE_INVALID`	Pass	Bad handle passed as a parameter.
C5D	Neg.	handle: 0 buffer: buf_ptr1 byte_count: 9000 offset: 9200	`SH_C_DSEG_OFFSET_BOUND`	Pass	The supplied offset parameter is outside the bounds of the 9,000 byte data segment.
C5E	Neg.	handle: 0 buffer: buf_ptr1 byte_count: 8000 offset: 2000	`SH_C_INSUFFICENT_SPACE`	Pass	The specified byte count 8,000 cannot be written within the bounds of 9,000 – 2,000 bytes.
C5F	Neg.	handle: 0 buffer: NULL	`SH_C_INPUT_BUFF_NULL`	Pass	Input buffer is null.

Table 18. Test group 5: `sh_c_write_dseg`.

Test group 6: `sh_c_read_dseg`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C6A	Pos.	handle: 0 buffer: buf_ptr1 byte_count: 9000 offset: 0	No error.	Pass	All 9,000 bytes of the data segment is read into buf_ptr1. Performed after C5A.
C6B.1	Pos.	handle: 1 buffer: buf_ptr2 byte_count: 4500 offset: 0	No error.	Pass	The first half of the data segment is read into buf_ptr2. Performed after C5B.1
C6B.2	Pos.	handle: 1 buffer: buf_ptr2 byte_count: 4500 offset: 4500	No error.	Pass	The second half of the data segment beginning at byte offset 4,500 is read into buf_ptr2. Performed after C5B.2
C6C	Neg.	handle: 2	`SH_C_HANDLE_INVALID`	Pass	Bad handle passed as a parameter.
C6D	Neg.	handle: 0 buffer: buf_ptr1 byte_count: 9000 offset: 9200	`SH_C_OFFSET_BOUND_ERROR`	Pass	The supplied offset parameter is outside the bounds of the 9,000 byte data segment.
C6E	Neg.	handle: 0 buffer: NULL	`SH_C_OUTPUT_BUFF_NULL`	Pass	Output buffer is null.

Table 19. Test group 6: `sh_c_read_dseg`.

Test group 7: `sh_c_close_dseg`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C7A.1	Pos.	handle: 0	No error, hash value generated and stored	Pass	Performed after C5A.
C7A.2	Neg.	handle: 0	`SH_C_HANDLE_INVALID`	Pass	Handle was already closed in test C7A.1.
C7B	Neg.	handle: 2	`SH_C_HANDLE_INVALID`	Pass	Bad handle passed.

Table 20. Test group 7: `sh_c_close_dseg`.

Test group 8: `sh_c_delete_dseg`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C8A.1	Pos.	path: “/” node: 1	No error, data segment deleted	Pass	/1 dseg was created prior to this test.
C8A.2	Neg.	path: “/” node: 1	`SH_C_NO_SUCH_NODE`	Pass	Data segment was already deleted.
C8B	Neg.	path: “/” node: 128	`SH_C_MAX_NODE_NAME`	Pass	Specified node name exceeds configured maximum of 128 nodes.
C8C	Neg.	path: “/” node: 0	`SH_C_NO_SUCH_NODE`	Pass	
C8D	Neg.	path: “/” node: 2	`SH_C_NODE_NOT_DSEG`	Pass	/2 directory was created prior to this test.
C8E	Neg.	path: “a”	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C8F	Neg.	path: “/a”	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C8G	Neg.	path: “/0”	`SH_C_PATH_ERROR`	Pass	Path does not exist

Table 21. Test group 8: `sh_c_delete_dseg`.

Table 22 contains the results for test group 9, the `sh_c_delete_subtree` interface. A simple tree structure was constructed to test the interface's ability to descend and remove data segment and directory table nodes from the specified parent node. Three directory table nodes with paths "/1", "/1/1", and "/1/2" were created along with two data segment nodes at paths "/1/1/1" and "/1/2/1".

Table 23 contains the results for test group 10, the `sh_c_get_child` interface. The tree structure constructed for test group 9 was also used for testing this interface's functionality.

Table 24 contains the results for test group 11, the `sh_c_get_dseg_size` interface. A data segment of size 9,000 bytes and path "/1" and a directory table node of path "/2" were created prior to testing. Table 25 contains the results fpr test group 12, the `sh_c_get_dseg_hash` interface. A data segment of size 9,000 bytes and path "/1" was created, opened, and written to with 9,000 bytes of pre-generated binary data. The hash of the data segment contents were calculated and stored in the test program. The data segment was then closed. A directory node of path "/2" was also created for negative testing purposes. The `sh_c_get_dseg_hash` interface was called and returned an equivalent hash, demonstrating the interface retrieved the correct hash value.

Table 26 contains the results for test group 13, the `sh_c_check_hash` interface. A data segment of size 9,000 bytes and path "/1" was created, opened, and written to with 9,000 bytes of pre-generated binary data. The data segment was then closed. A directory node of path "/2" was also created for negative testing purposes.

Test group 9: `sh_c_delete_subtree`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C9A	Neg.	path: “/1/1” node: 1	`SH_C_NODE_NOT_DIRNODE`	Pass	The specified node was a data segment.
C9B.1	Pos.	path: “/” node: 1	No error, directory and child nodes deleted.	Pass	A hierarchy off of /1 was created prior to this test.
C9B.2	Neg.	path: “/” node: 1	`SH_C_NO_SUCH_NODE`	Pass	Directory table node was already deleted.
C9B.3	Neg.	path: “/1” node: 1	`SH_C_PATH_ERROR`	Pass	This path no longer exists because parent node was deleted in test C9A.1
C9C	Neg.	path: “/” node: 128	`SH_C_MAX_NODE_NAME`	Pass	Node name exceeds configured maximum of 128 nodes.
C9D	Neg.	path: “/” node: 0	`SH_C_NO_SUCH_NODE`	Pass	No node exists at path “/0”
C9E	Neg.	path: “a”	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C9F	Neg.	path: “/a”	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C9G	Neg.	path: “/0”	`SH_C_PATH_ERROR`	Pass	Path does not exist

Table 22. Test group 9: sh_c_delete_subtree.

Test group 10: `sh_c_get_child`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C10A	Pos.	path: “/” node: 1 offset: 0	No error, node name and directory node type returned for node at path “/1/1”	Pass	
C10B	Pos.	path: “/” node: 1 offset: 1	No error, node name and directory node type returned for node at path “/1/2”	Pass	
C10C	Neg.	path: “/” node: 1 offset: 2	`SH_C_OFFSET_BOUND_ERROR`	Pass	
C10D	Pos.	path: “/1” node: 1 offset: 0	No error, node name and data segment node type returned for node at path “/1/1/1”	Pass	
C10E	Neg.	path: “/1/1” node: 1	`SH_C_NODE_NOT_DSEG`	Pass	The specified node is a data segment.
C10F	Neg.	path: “/” node: 128	`SH_C_MAX_NODE_NAME`	Pass	Node name exceeds configured maximum of 128 nodes.
C10G	Neg.	path: “/” node: 0	`SH_C_NO_SUCH_NODE`	Pass	No node exists at path “/0”
C10H	Neg.	path: “a”	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C10I	Neg.	path: “/a”	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C10J	Neg.	path: “/0”	`SH_C_PATH_ERROR`	Pass	Path does not exist

Table 23. Test group 10: `sh_c_get_child`.

Test group 11: `sh_c_get_dseg_size`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C11A	Pos.	path: “/” node: 1	No error, data segment size returned	Pass	/1 dseg created prior to this test.
C11B	Neg.	path: “/” node: 2	`SH_C_NODE_NOT_DSEG`	Pass	/2 was created as a directory node prior to this test.
C11C	Neg.	path: “/” node: 3	`SH_C_NO_SUCH_NODE`	Pass	No node exists at path “/3”
C11D	Neg.	path: “/” node: 128	`SH_C_MAX_NODE_NAME`	Pass	Node name exceeds configured maximum of 128 nodes.
C11E	Neg.	path: “a”	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C11F	Neg.	path: “/a”	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C11G	Neg.	path: “/0”	`SH_C_PATH_ERROR`	Pass	Path does not exist

Table 24. Test group 11: sh_c_get_dseg_size.

Test group 12: `sh_c_get_dseg_hash`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C12A	Pos.	path: "/" node: 1	No error, data segment hash returned	Pass	/1 dseg was created prior to this test.
C12B	Neg.	path: "/" node: 2	`SH_C_NODE_NOT_DSEG`	Pass	/2 was created as a directory node prior to this test.
C12C	Neg.	path: "/" node: 3	`SH_C_NO_SUCH_NODE`	Pass	No node exists at path "/3"
C12D	Neg.	path: "/" node: 128	`SH_C_MAX_NODE_NAME`	Pass	Node name exceeds configured maximum of 128 nodes.
C12E	Neg.	path: "a"	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C12F	Neg.	path: "/a"	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C12G	Neg.	path: "/0"	`SH_C_PATH_ERROR`	Pass	Path does not exist

Table 25. Test group 12: sh_c_get_dseg_hash.

Test group 13: `sh_c_check_hash`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
C13A	Pos.	path: “/” node: 1	No error, true value returned	Pass	/1 dseg was created prior to this test
C13B	Neg.	path: “/” node: 2	`SH_C_NODE_NOT_DSEG`	Pass	/2 was created as a directory node prior to this test.
C13C	Neg.	path: “/” node: 3	`SH_C_NO_SUCH_NODE`	Pass	No node exists at path “/3”
C13D	Neg.	path: “/” node: 128	`SH_C_MAX_NODE_NAME`	Pass	Node name exceeds configured maximum of 128 nodes.
C13E	Neg.	path: “a”	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C13F	Neg.	path: “/a”	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
C13G	Neg.	path: “/0”	`SH_C_PATH_ERROR`	Pass	Path does not exist

Table 26. Test group 13: `sh_c_check_hash`.

2. Initialization Interface Testing

Initialization interface testing is only fully informative if it occurs after data segment and directory nodes have already been created. All tests were performed after the test storage hierarchy was modified as described for each group.

Table 27 contains the results for test group 1, the `sh_i_initialize_databases` interface. Table 28 contains the results for test group 2, the `sh_i_get_handle` interface. The storage hierarchy was initialized with a handle table size of one row and two data segment of paths "/1" and "/2" were created prior to this test. A directory table node was also created at path "/3". Table 29 contains the results for test group 3, the `sh_i_read_in` interface. A data segment at size 9,000 bytes and path "/1" was created, opened, and written to with 9,000 bytes of pre-generated data using configuration interfaces prior to this test. The interface was called and the expected data was loaded into a test program buffer.

Table 30 contains the results for test group 4, the `sh_i_get_dseg_size` interface. A data segment of size 9,000 bytes and path "/1" was created prior to testing. Table 31 contains the results for test group 5, the `sh_i_get_dseg_hash` interface. A data segment of size 9,000 bytes and path "/1" was created, opened, and written to with 9,000 bytes of pre-generated binary data using configuration interfaces. The hash of the data segment contents were calculated and stored in the test program. The data segment was then closed. A directory node at path "/2" was also created for negative testing purposes. The `sh_i_get_dseg_hash` interface was called and returned the same hash value, demonstrating the interface retrieved the correct hash value.

Test group 1: `sh_i_initialize_databases`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
I1A	Pos.	block_offset: 8	No error, database structures loaded.	Pass	Offset 8 is the correct value
I1B	Neg.	block_offset: 0	`SH_C_CONFIG_DB_LOAD_ERR`	Pass	
I14C	Neg.	block_offset: 8	`SH_C_CLUS_DB_LOAD_ERR`	Pass	Configuration database altered to reflect incorrect location of cluster database.
I1D	Neg.	block_offset: 8	`SH_C_SAT_LOAD_ERR`	Pass	Configuration database altered to reflect incorrect location of SAT

Table 27. Test group 1: sh_i_initilizase_databases.

Test group 2: `sh_i_get_handle`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
I2A.1	Pos.	path: "/" node: 1	No error, handle zero returned.	Pass	/1 dseg was created prior to this test.
I2A.2	Neg.	path: "/" node: 2	`SH_C_HANDLE_TABLE_FULL`	Pass	All handles were consumed prior to this test.
I2B	Neg.	path: "/" node: 3	`SH_C_NODE_NOT_DSEG`	Pass	/3 directory node was created prior to this test.
I2C	Neg.	path: "/" node: 10	`SH_C_NO_SUCH_NODE`	Pass	No node exists at path "/10"
I2D	Neg.	path: "/" node: 128	`SH_C_MAX_NODE_NAME`	Pass	Node name exceeds configured maximum of 128 nodes.
I2E	Neg.	path: "a"	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
I2F	Neg.	path: "/a"	`SH_C_MALFORMED_PATH`	Pass	Input path is malformed.
I2G	Neg.	path: "/0"	`SH_C_PATH_ERROR`	Pass	Path does not exist

Table 28. Test group 2: `sh_i_get_handle`.

Test group 3: `sh_i_read_in`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
I3A	Pos.	handle: 0 buffer: buf_ptr1	No error, data read in from data segment.	Pass	A valid handle was obtained prior to this test.
I3B	Neg.	handle: 1	`SH_C_HANDLE_INVALID`	Pass	Bad handle passed.

Table 29. Test group 3: `sh_i_read_in`.

Test group 4: `sh_i_get_dseg_size`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
I4A	Pos.	handle: 0	No error, data segment size returned	Pass	A valid handle was obtained prior to this test.
I4B	Neg.	handle: 1	`SH_C_HANDLE_INVALID`	Pass	Bad handle passed.

Table 30. Test group 4: sh_i_get_dseg_size.

Test group 5: `sh_i_get_dseg_hash`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
I5A	Pos.	handle: 0	No error, data segment hash returned	Pass	A valid handle was obtained prior to this test.
I5B	Neg.	handle: 1	`SH_C_HANDLE_INVALID`	Pass	Bad handle passed.

Table 31. Test group 5: sh_i_get_dseg_hash.

3. Runtime Interface Testing

Storage hierarchy runtime interfaces must be called after the handle table database has been initialized and handles have been associated with data segments. Each runtime interface test group is performed after these interfaces have been called and one data segment with handle zero is has been retrieved by the `sh_i_get_handle` interface call. This data segment is of size 9,000 bytes and contains pre-generated binary data. The test program calculates and stores the hash value of this data for future comparison.

Table 32 contains the results for test group 1, the `sh_r_read_in` interface. The interface is called using a handle value `of 0` and stores the data read from the data segment in a test program memory buffer. Table 33 contains the results for test group 2, the `sh_r_write_out` interface, which is performed immediately after test group 1. The test program modifies the 9,000 byte buffer in memory, calculates the new hash value, and calls the interface to write the data back to the data segment. Table 34 contains the results for test group 3, the `sh_r_get_dseg_hash` interface. This test is performed immediately after test group 2, and returns the hash generated by the `sh_r_write_out` interface. This hash is compared with the hash generated by the test program in test group 2. By demonstrating that the two hashes are equal, both `sh_r_write_out` and `sh_r_get_dseg_hash` interfaces are shown to function correctly. The `sh_r_write_out` interface generates the correct hash value, which is returned by calling the `sh_r_get_dseg_hash` interface.

Table 35 contains the results for test group 4, the `sh_r_get_dseg_size` interface. Table 36 contains the results for test group 5, the `sh_r_check_hash` interface. A data segment of size 9,000 bytes and path “/1” was created, opened, and written to with 9,000 bytes of pre-generated binary data. The data segment was then closed.

Test group 1: `sh_r_read_in`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
R1A	Pos.	handle: 0 buffer: buf_ptr1	No error, data read in from data segment.	Pass	A valid handle was obtained prior to this test.
R1B	Neg.	handle: 1	`SH_C_HANDLE_INVALID`	Pass	Bad handle passed.

Table 32. Test group 1: `sh_r_read_in`.

Test group 2: `sh_r_write_out`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
R2A	Pos.	handle: 0 buffer: buf_ptr1	No error, data written to data segment and new hash is generated.	Pass	A valid handle was obtained prior to this test.
R2B	Neg.	handle: 1	`SH_C_HANDLE_INVALID`	Pass	Bad handle passed.

Table 33. Test group 2: `sh_r_write_out`.

Test group 3: `sh_r_get_dseg_hash`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
R3A	Pos.	handle: 0	No error, data segment hash returned	Pass	A valid handle was obtained prior to this test.
R3B	Neg.	handle: 1	`SH_C_HANDLE_INVALID`	Pass	Bad handle passed.

Table 34. Test group 3: sh_r_get_dseg_hash.

Test group 4: `sh_r_get_dseg_size`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
R4A	Pos.	handle: 0	No error, data segment size returned	Pass	A valid handle was obtained prior to this test.
R4B	Neg.	handle: 1	`SH_C_HANDLE_INVALID`	Pass	Bad handle passed.

Table 35. Test group 4: sh_r_get_dseg_size.

Test group 5: `sh_r_check_hash`					
Test	**Type**	**Parameters**	**Expected Result**	**Pass?**	**Remarks**
R5A	Pos.	handle: 0	No error, true value returned	Pass	A valid handle was obtained prior to this test.
R5B	Neg.	handle: 1	`SH_C_HANDLE_INVALID`	Pass	Bad handle passed.

Table 36. Test group 5: `sh_r_check_hash`.

C. SUMMARY

This chapter presented a storage hierarchy scenario and the results of tests performed on each external interface. The procedures for recreating these test results are contained in Appendix B. Chapter V discusses the results of this study, related work, and suggests future work to improve upon the results of this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

V. RESULTS

The storage hierarchy prototype developed in this thesis is a near-complete implementation of the LPSK product functional specification for secondary storage of data segments [15]. The design and development of the prototype has shown that requirements can be derived from this specification for secondary storage of data segments, leading to the design, implementation, and successful testing of a storage hierarchy prototype. The prototype presents interfaces that can be used to construct a storage hierarchy structure on storage media for which a hardware abstraction layer has been provided. The prototype contains modules that are used for off-line configuration and both LPSK initialization and runtime operation modes to access the contents of data segments. The prototype is capable of maintaining configuration and data segment permanence by saving its configuration databases to secondary storage and loading the configuration databases at some later time. This storage hierarchy prototype, however, is not suitable for inclusion in the LPSK binary due to several incomplete features and a prototype design choice. These issues are discussed in this chapter. The chapter starts with a discussion of several problems and challenges encountered in this study and continues with a presentation of incomplete features of the storage hierarchy prototype, related work, and suggestions for future work related to the LPSK storage hierarchy.

A. PROBLEMS ENCOUNTERED

1. Hardware Abstraction

The first challenge encountered was the lack of LPSK disk device driver software. This driver software facilitates the reading and writing of data between programs and a particular secondary storage device, such as a hard disk drive. Without this software layer, many aspects of the storage hierarchy prototype that rely on storing and modifying configuration information could not be tested directly on the LPSK and shown to function correctly. The solution to this problem was to design a hardware abstraction layer to hide the complexity of reading and writing to disk from the storage hierarchy interfaces. The linux *dd* command-line tool was used by a HAL developed in this thesis to read and write data to a virtual disk, as described in the implementation methodology found in Chapter III.

2. Memory Allocation

Another challenge was the lack of the complete set of standard C library functions on the LPSK platform. Program developers rely on C libraries to perform oftentimes complex programming tasks quickly and easily, such as reserving primary memory or working with C strings. Implementation of functions that were necessary for storage hierarchy modules but had not been implemented for the LPSK added to the overall programming workload. This challenge led to an additional problem regarding memory allocation.

The handle table and SAT databases vary in size based on the configuration of storage hierarchy. Primary memory is reserved for these databases dynamically during their initialization. In the context of the configuration interfaces, this can be accomplished by the C standard library call `malloc`, which returns a pointer to a primary memory data segment of the specified size. This library call interacts with the operating system's memory manager to reserve this memory. A comparable library call does not exist in the LPSK platform. The LPSK memory model is also fundamentally dissimilar to the memory model of the Linux testing workstation. Because all aspects of the storage hierarchy were tested on a Linux platform, it was necessary to substitute portions of code that would have otherwise interacted with the LPSK memory manager with calls to the C standard library `malloc` Dynamic memory allocation was also avoided in situations where it might otherwise have led to more streamlined and elegant code.

3. File System Analogy

This study often relied on the similarities found in the functional specification for the secondary storage of LPSK data segments and the structure of contemporary file systems. The storage hierarchy organizes data segments in a tree-like hierarchal fashion. This is similar to the way file systems such as Microsoft Corporation's FAT file system or the Linux “ext3” file system organize files in a hierarchal directory-tree structure. The contemporary file system analogy was attractive because of the way it framed the discussion of the requirements and design of the storage hierarchy prototype in context likely to be familiar to readers. This approach of treating the storage hierarchy like a specialized file system, however, influenced the design of the storage hierarchy prototype in such a way that the design was no longer congruent to the functional specification. It was assumed that an inner node—a node that has child nodes—could not also be a data segment. While the LPSK functional specification does not explicitly require that inner nodes may represent data segments, the assumption arrived at in the design of this storage hierarchy prototype does not allow such dual-purpose nodes. As a result, the storage hierarchy prototype developed in this thesis may not be said to be a complete or fully accurate LPSK storage hierarchy implementation, depending on the interpretation of the functional specification. Significant modification of the storage hierarchy prototytpe would be required to allow inner nodes to contain data..

B. INCOMPLETE FEATURES

Several features of the LPSK storage hierarchy prototype are incomplete due to a lack of documented requirements or incomplete features of the LPSK itself. The prototype is not capable of making the LPSK system calls necessary to reserve dynamically-sized segments of primary memory. This functionality must be added to the prototype to make it capable of executing as part of the LPSK binary.

Many initialization and runtime interfaces are required to cause the LPSK platform to halt on certain error cases. Comment lines are included in storage hierarchy module files indicating where the system calls to make the kernel halt are to be inserted.

Another incomplete feature involves data segment MAC generation. The external interface specification developed in this thesis does not include parameters that allow the interface caller to supply the key used to generate a data segment MAC. Instead, the storage hierarchy prototype makes use of a SHA-256 hash implementation to generate a collision-free substitution MAC used for testing the storage hierarchy external interfaces.

The prototype can be modified to accept MAC keys through the external interfaces in both LPSK and non-LPSK contexts, or in the case of LPSK context interfaces, code can be added to access the primary memory location of the MAC key.

C. RELATED WORK

This section presents several related data and file storage systems.

Security Enhanced Linux (SELinux) is a joint effort between the National Security Agency (NSA) and the Linux community to implement a variety of mandatory access control policies for the Linux operating system [19]. It associates security labels with file system objects and users, and mediates access to files in the Linux file system based on SELinux security policies. SELinux policies separate information based on confidentiality and integrity labels. SELinux components must be built into and compiled with the Linux kernel in order to enforce its policies. Since SELinux functionality is bolted on to the Linux kernel, which is not a high assurance operating system, it does not meet the high assurance objectives of the LPSK.

The XTS-400 Trusted Computer System is a commercial high assurance computing platform combining evaluated hardware and the STOP operating system [20]. XTS-400 systems host and separate users and information of various sensitivity levels and stores data in a hierarchical file system. It has been successfully evaluated at EAL5+ [21]

The Gemini Trusted Network Processor (GTNP), a product of Gemini Computers, Inc., implements a specialized storage mechanism for the secondary storage of its data segments [22]. A GTNP data segment is a variable sized unit of storage that can be swapped in and out of primary memory to some secondary store by a segment storage system. All GTNP processes to which a GTNP data segment has made known modify the same segment in primary memory. Neither the STOP OS nor the GTNP are separation kernels and reflect an architectural approach different than that of the LPSK.

Like the LPSK, GTNP data segments are logically organized in a hieratical fashion. Every node of the hierarchy can simultaneously contain data and have a number of child nodes. Data segments are made known to processes by exporting a path to a subtree. If there is a hierarchy of GTNP data segments labeled 5-6-9-10, for example, and the path 5-6 is exported to a process, then data segments 9 and 10 have been made known to the process. The process is unaware of any data segments above this subtree.

D. FUTURE WORK

This section presents suggestions for future work that can be used to improve the storage hierarchy prototype developed in this thesis.

1. Performance and Optimization

The main factor influencing the performance of the storage hierarchy external interfaces, in terms of execution time, is storage medium access. Interfaces access to the storage medium through the HAL. Every interface accesses the storage medium to some varying degree, with some interfaces performing ten or more discrete disk access events. As interfaces traverse the tree structure of particularly deep storage hierarchy trees, the number of discrete accesses can rapidly increase. One optimization that could be employed to reduce the amount of execution time spent waiting for disk accesses to complete is to cache the commonly accessed disk sectors.

Directory table clusters are read from the storage medium by any interface that takes a storage hierarchy path as a parameter. These directory tables are loaded into memory and are searched for the SAT index of the next directory table of the path, which is then loaded into memory, until the entire path has been traversed by this cycle of directory table reference. The principle of temporal locality [23] suggests that if a directory table has been recently accessed, it will be accessed again in the near future. During configuration, for example, after the interface call has been made to create a directory table at a given path, it is likely that future interface calls will use the same path to populate the newly created directory table with child data segment and directory table nodes. A directory table caching module would reduce the execution time of many interfaces by caching directory table structures in memory and intercepting calls to the HAL to load a directory table if it already exists in the cache.

Each storage hierarchy configuration interface acts as an atomic operation. Any modifications that are made to the SAT, directory tables, or cluster database are immediately committed to the storage medium by the HAL. While this behavior has positive attributes—a storage hierarchy remains in a valid state if a configuration tool unexpectedly ends execution—it also has some negative performance attributes in that the storage medium is accessed often. A configuration interface and supporting module could be added to the prototype that only commits the modified databases to the storage medium after the configuration tool user has decided to end a configuration session. This feature would reduce the amount of storage medium access during configuration.

2. Formatting Tool

A storage hierarchy's storage medium must be formatted with information about the configuration and dimensions of the storage hierarchy before configuration interfaces may be used. This was accomplished during testing of the storage hierarchy external interfaces by a C program that created and populated the basic storage hierarchy structures in memory and then invoked the HAL to commit these structures to the disk. Chapter IV contains a description of how this was accomplished by the test program. The test program, however, relied on hard-coded values when formatting the storage medium. Configuration interfaces could be added to the storage hierarchy prototype that would allow a configuration tool to format a storage medium using the parameters it specifies.

E. CONCLUSION

The storage hierarchy prototype was designed and implemented to meet the requirements derived from the LPSK product functional specification for the secondary storage of data segments. While the prototype is not a wholly accurate implementation, its development has increased the attention paid to this aspect of the LPSK and has generated documentation and C code that can be used to further the goals of the TCX project. This study began with the analysis of the LPSK functional specifications and continued with the creation of an external interface specification document which demarcated the boundary between the storage hierarchy and the rest of the LPSK (see Appendix A). Development began after internal databases and modules of the storage hierarchy had been identified. Development was followed by a series of tests and prototype corrections to confirm the positive and negative behaviors of the external interfaces conformed to the external interface specification.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: LPSK STORAGE HIEARCHY INTERFACES

This appendix outlines the external interfaces of the storage hierarchy. Interfaces are groups of configuration, initialization, and runtime functions and are detailed in separate sections. Constants referenced in the following sections are defined in the respective header files of each function group. Functions return a success code NO_ERROR if the function call is successful.

For the purposes of Appendix A, primitive variables referred to as int (integer) are of 32-bit length.

A. CONFIGURATION INTERFACE

This section describes the functions used to configure the storage hierarchy executing in a non-LPSK context. The functions are implemented and executed on a system that has access to the C programming language standard libraries, including standard input/output. The functions are exposed to a configuration tool as a series of function calls. They are:

- sh_c_initialize_databases
- sh_c_make_dseg
- sh_c_make_subtree
- sh_c_open_dseg
- sh_c_write_dseg
- sh_c_close_dseg
- sh_c_delete_dseg
- sh_c_delete_subtree
- sh_c_get_child
- sh_c_get_dseg_size
- sh_c_get_dseg_hash
- sh_c_read_dseg
- sh_c_check_hash

The following subsection details these functions.

1. `sh_c_initialize_databases`

This function reads the storage hierarchy configuration from disk into memory.

1.1 Prototype

```
unsigned int sh_c_initialize_databases
    const unsigned int offset);
```

1.2 Inputs

- offset
  The disk block offset into the storage medium where the storage hierarchy configuration data is located.

1.3 Outputs

- Function Result
  A numerical value that indicates success or failure of the function.

1.4 Processing

1. Attempt to load the configuration database into memory from disk at input offset. If a configuration database is not found, return SH_C_CONFIG_DB_LOAD_ERR error. If memory can not be allocated, return SH_C_MEM_ALLOC_ERR error.
2. Attempt to load the cluster database into memory from disk. If a cluster database is not found, return SH_C_CLUS_DB_LOAD_ERR error. If memory can not be allocated, return SH_C_MEM_ALLOC_ERR error.
3. Attempt to load the SAT into memory from disk. If a SAT structure is not found, return SH_C_SAT_LOAD_ERR. If memory can not be allocated, return SH_C_MEM_ALLOC_ERR error.
4. Construct the handle to data segment table in memory.
5. Return NO_ERROR.

1.5 Effects

- This function reduces the available primary memory to accommodate the databases.
-

1.6 Errors

SH_C_CONFIG_DB_LOAD_ERR

This error is returned if the data loaded from disk does not represent a configuration database structure.

SH_C_CLUS_DB_LOAD_ERR

> This error is returned if the data loaded from disk does not represent a cluster database structure.

SH_C_SAT_LOAD_ERR

> This error is returned if the data loaded from disk does not represent a SAT structure.

SH_C_MEM_ALLOC_ERR

> This error is returned if a memory allocation library call fails to return a valid buffer pointer.

2. `sh_c_make_dseg`

This function creates a child node in the storage hierarchy and designates it as a data segment node.

2.1 Prototype

```
unsigned int sh_c_make_dseg(
    const char * const path
    const unsigned int node
    const unsigned int size);
```

2.2 Inputs

- path
  The absolute path in the storage hierarchy of the parent of the new child node. The string is null-terminated and cannot exceed MAX_PATH_LENGTH bytes in length (including the null terminator).
- node
  The name of the new data segment node.
- size
  Defines the size of the data segment in bytes.

2.3 Outputs

- Function Result
  A numerical value that indicates success or failure of the function.

2.4 Processing

1. Verify that there exists enough free disk space to create a data segment of the specified input size. If the space does not exist, then return SH_C_DISKSPACE error.
2. Validate the input path as a correctly formed path string. Return SH_C_MALFORMED_PATH error if validation fails. The path shall only consist of the following characters: /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, NULL.
3. Attempt to walk the storage hierarchy along the specified input path. Return SH_C_PATH_ERROR error if the path specified by the input path does not exist in the storage hierarchy.
4. Verify that the specified child node to be created does not already exist. If it does, return the SH_C_NODE_ALREADY_EXISTS error.
5. Verify that the addition of this child would not exceed the configured maximum child limit. Return SH_C_CHILD_LIMIT_REACHED error of it does.
6. Instantiate the data segment node by updating the parent node's node table structure, reserve required storage memory, initialize the storage memory to 0x0.

7. Calculate the message authenticate code for the segment and store the hash in the parent node's table structure.
8. Return NO_ERROR.

2.5 Effects

- The node directory table of the new data segment node's parent node is updated to reflect the creation of the new data segment node.
- Space necessary to store the data segment is reserved by updating the internal data structures of the storage hierarchy.
- The data segment node entry in the directory table structure is modified, with the data segment length field set to the specified input size and the hash field set to the calculated hash.
- The data segment is initialized to 0x0 in every byte.

2.6 Errors

SH_C_DISKSPACE

This error is returned if there is not enough free disk space to create a data segment of the input size.

SH_C_MALFORMED_PATH

This error is returned if the input path is not a correctly formed concatenated string of integer values separated by forward-slashes, "/."

SH_C_PATH_ERROR

This error is returned if the input path does not exist in the storage hierarchy tree structure.

SH_C_NODE_ALREADY_EXISTS

This error is returned if the node specified by the input path and node already exists.

SH_C_CHILD_LIMIT_REACHED

This error is returned if creating a new child node would exceed the configured maximum child count at the level specified by input path.

3. `sh_c_make_subtree`

This function creates a child node in the storage hierarchy and designates it as a directory node.

3.1 Prototype

```
unsigned int sh_c_make_subtree(
    const char *  const path
    const unsigned int node);
```

3.2 Inputs

- path
  The absolute path in the storage hierarchy to the parent of a new child node. The string is null-terminated and cannot exceed MAX_PATH_LENGTH bytes in length.
- node
  The name of the new directory node.

3.3 Outputs

- Function Result
  A numerical value that indicates success or failure of the function on the specified path.

3.4 Processing

1. Verify that there exists enough free disk space to create a directory node If the space does not exist, then return SH_C_DISKSPACE error.
2. Validate the input path as a correctly formed path string. Return SH_C_MALFORMED_PATH error if validation fails. The path shall only consist of the following characters: /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, NULL.
3. Attempt to walk the storage hierarchy along the specified input path. Return SH_C_PATH_ERROR error if the path specified by the input path does not exist in the storage hierarchy.
4. Verify that the specified child node to be created does not already exist. If it does, return the SH_C_NODE_ALREADY_EXISTS error.
5. Allocate disk space for the node and initialize it to 0x0.
6. Instantiate the directory node by updating the parent node's node directory structure
7. Return NO_ERROR.

3.5 Effects

- The node directory table of the new directory node's parent node is updated to reflect the creation of the new directory node.
- A new directory table structure is created in a free data block for the new directory node's table structure.
- The amount of available disk space is reduced

3.6 Errors

SH_C_DISKSPACE

This error is returned if there is not enough free disk space to create a directory node.

SH_C_MALFORMED_PATH

This error is returned if the input path is not a correctly formed concatenated string of integer values separated by forward-slashes, "/."

SH_C_PATH_ERROR

This error is returned if the input path does not exist in the storage hierarchy tree structure.

SH_C_NODE_ALREADY_EXISTS

This error is returned if the node specified by the input path already exists.

4. `sh_c_open_dseg`

This function returns a handle associated with the specified data segment and allows it to be written to and read from.

4.1 Prototype

```
unsigned int sh_open_dseg(
        const char * const path
        const unsigned int node
        const unsigned int * handle);
```

4.2 Inputs

- path
  The absolute path in the storage hierarchy to the parent of a data segment node. The string is null-terminated and cannot exceed MAX_PATH_LENGTH bytes in length.
- node
  The name of the data segment node.

4.3 Outputs

- Handle
  A handle to the associated data segment.
- Function Result
  A numerical value that indicates success or failure of the function.

4.4 Processing

1. Verify that there is an available handle to return. Return SH_C_HANDLE_TABLE_FULL if no handle is available.
2. Validate the input path as a correctly formed path string. Return SH_C_MALFORMED_PATH error if validation fails. The path shall only consist of the following characters: /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, NULL.
3. Attempt to walk the storage hierarchy along the input path. Return SH_C_PATH_ERROR error if the path specified by the input path does not exist in the storage hierarchy.
4. Verify that there exists a child at the input node off the parent. If no such child node exists, return SH_C_NO_SUCH_NODE.
5. Verify that the specified child node is a data segment node. If it is a directory node, return SH_C_NODE_NOT_DSEG error.
6. Verify that the specified data segment is not already in an open state. Return SH_C_DSEG_ARLEADY_OPEN error if the segment is already open.
7. Associate a handle with the data segment and return the handle.
8. Return NO_ERROR.

4.5 Effects

- The configuration interface open data segment database is modified to associate the generated handle value with the specified data segment. The data segment is designated as being in an ‘open’ state.

4.6 Errors

SH_C_HANDLE_TABLE_FULL

> This error is returned if too many segments are already open and no handles are available.

SH_C_MALFORMED_PATH

> This error is returned if the input path is not a correctly formed concatenated string of integer values separated by forward-slashes, “/.”

SH_C_PATH_ERROR

> This error is returned if the input path does not exist in the storage hierarchy tree structure.

SH_C_NO_SUCH_NODE

> This error is returned if the combination of input path and node does not exist.

SH_C_NODE_NOT_DSEG

> This error is returned if the node specified by the input path is not designated as a data segment node.

SH_C_DSEG_ALREADY_OPEN

> This error is returned if the specified data segment is already open.

5. sh_c_write_dseg

This function writes data from the specified buffer to the specified data segment starting at the offset provided.

5.1 Prototype

```
unsigned int sh_c_write_dseg(
        const unsigned int handle
        const void * const buffer
        const unsigned int byte_count
        const unsigned int offset
        unsigned int * bytes_written);
```

5.2 Inputs

- handle
  The handle to the data segment to be written to.
- buffer
  The buffer from which the data is read.
- byte_count
  The number of bytes to write into the data segment.
- offset
  The offset into the data segment where writing is to begin.

5.3 Outputs

- bytes_written
  The actual number of bytes written to the data segment.
- Function Result
  A numerical value that indicates success or failure of the function.

5.4 Processing

1. Set output bytes written to zero.
2. Validate the input handle is associated with an open segment. Return SH_C_HANDLE_INVALID if it is not
3. Verify that the specified offset is within the bounds of the specified data segment. Return SH_C_OFFSET_BOUND_ERROR error if it is not.
4. Verify that the input buffer is not NULL. Return SH_C_INPUT_BUFF_NULL if it is null.
5. Verify that the specified number of bytes to read from the buffer can be accommodated by the data segment from the specified offset. Return SH_C_INSUFFICENT_SPACE error if it cannot.
6. Attempt to write the specified number of bytes from the specified buffer. Set the output bytes written to the number of bytes successfully written.
7. Return NO_ERROR.

5.5 Effects

- Disk sectors are modified. If a close operation is not performed before the system halts, then the segment will be in a corrupted state because the stored hash will not correspond to the contents of the segment.
- If the input buffer is an invalid pointer, the behavior of the program is undefined, and will probably lead to a termination of the program by the operating systems.

5.6 Errors

SH_C_HANDLE_INVALD

This error if the specified handle is not a valid handle because it does not exist in the open data segment database.

SH_C_OFFSET_BOUND_ERROR

This error is returned if the specified input offset is outside the bounds of the data segment.

SH_C_INPUT_BUFF_NULL

This error is returned if the specified input buffer pointer is NULL.

SH_C_INSUFFICENT_SPACE

This error is returned if writing the amount of bytes specified by input length into the specified data segment beginning at the specified input offset would cause data to be written beyond the edge of the data segment.

6. `sh_c_close_dseg`

This function closes the data segment associated with the supplied handle and calculates and stores the hash value for the segment.

6.1 Prototype

unsigned int sh_close_dseg(const unsigned int handle);

6.2 Inputs

- handle
  The handle of a data segment to be closed.

6.3 Outputs

- Function Result
  A numerical value that indicates success or failure of the function.

6.4 Processing

1. Validate the input handle is associated with a data segment. Return SH_C_HANDLE_INVALID error if it is not.
2. Calculate the hash value of the data stored in the data segment associated with the input handle and update the meta-data for the segment in the parent directory node.
3. Remove the entry associated with the specified handle from the open data segment database node.
4. Return NO_ERROR.

6.5 Effects

- The hash value stored in the meta-data associated with the data segment is updated to reflect its (potentially) new contents. The handle is removed from the open data segment database and the data segment may no longer be written to or read from until it is opened again.

6.6 Errors

SH_C_HANDLE_INVALD

This error if the specified handle is not a valid handle because it does not exist in the open data segment database.

7. `sh_c_delete_dseg`

This function removes a data segment node from the storage hierarchy and frees its associated data blocks.

7.1 Prototype

```
unsigned int sh_c_delete_dseg(
    const char * const path
    const unsigned int node);
```

7.2 Inputs

- path
  The absolute path in the storage hierarchy to the parent of the data segment to be deleted. The string is null-terminated and cannot exceed MAX_PATH_LENGTH bytes in length.
- node
  The name of the node to be deleted.

7.3 Outputs

- Function Result
  A numerical value that indicates success or failure of the function.

7.4 Processing

1. Validate the input path as a correctly formed path string. Return SH_C_MALFORMED_PATH error if validation fails. The path shall only consist of the following characters: /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, NULL.
2. Attempt to walk the storage hierarchy along the specified input path. Return SH_C_PATH_ERROR error if the path specified by the input path does not exist in the storage hierarchy.
3. Verify that a data segment record exists for the input node and path. Return SH_C_NO_SUCH_NODE if it does not.
4. Verify that the specified child node is a data segment node. If it is a directory node, return SH_C_NODE_NOT_DSEG error.
5. Remove the data segment node's meta-data from the storage hierarchy and free the storage memory associated with the deleted node.
6. Return NO_ERROR.

7.5 Effects

- The parent of the deleted node is updated to reflect the removal of the data segment node.

- Storage hierarchy internal data structures are modified so that storage memory allocated to the data segment are freed and available for future allocation.

7.6 Errors

SH_C_MALFORMED_PATH

This error is returned if the input path is not a correctly formed concatenated string of integer values separated by forward-slashes, "/."

SH_C_PATH_ERROR

This error is returned if the input path does not exist in the storage hierarchy tree structure.

SH_C_NO_SUCH_NODE

This error is returned if the combination of input path and node does not exist.

SH_C_NODE_NOT_DSEG

This error is returned if the node specified by the input path is not designated as a data segment node.

8. `sh_c_delete_subtree`

This function removes the specified directory node and all of its children nodes from the storage hierarchy.

8.1 Prototype

```
unsigned int sh_c_delete_subtree(
    const char * const path
    const unsigned int node);
```

8.2 Inputs

- path
  The absolute path in the storage hierarchy to the parent of the directory node to be removed. The string is null-terminated and cannot exceed MAX_PATH_LENGTH bytes in length.
- node
  The name of the directory table node.

8.3 Outputs

- Function Result
  A numerical value that indicates success or failure of the function.

8.4 Processing

1. Validate the input path as a correctly formed path string. Return SH_C_MALFORMED_PATH error if validation fails. The path shall only consist of the following characters: /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, NULL
2. Attempt to walk the storage hierarchy along the specified input path. Return SH_C_PATH_ERROR error if the path specified by the input path does not exist in the storage hierarchy or only consists of '/'.
3. Verify that a directory table record exists for the input node and path. Return SH_C_NO_SUCH_NODE if it does not
4. Verify that the specified node is a directory node. If it is a data segment node, return SH_C_NODE_NOT_DNODE error.
5. For all child data segment nodes, call `sh_c_delete_dseg`. For all child directory nodes, call this function (recursive).
6. Remove the meta-data of the input node from the directory table associated with the input path.
7. Mark the storage memory associated directory the node as available.
8. Return NO_ERROR.

8.5 Effects

- The node directory table associated with the input path is updated to reflect the removal of the input node.
- Storage hierarchy internal data structures are modified so that storage memory allocated to the directory node and all of its children nodes are freed and available for future allocation.

8.6 Errors

SH_C_MALFORMED_PATH

This error is returned if the input path is not a correctly formed concatenated string of integer values separated by forward-slashes, "/."

SH_C_PATH_ERROR

This error is returned if the input path does not exist in the storage hierarchy tree structure.

SH_C_NO_SUCH_NODE

This error is returned if the combination of input path and node does not exist.

SH_C_NODE_NOT_DNODE

This error is returned if the node specified by the input path is not designated as a directory node.

9. `sh_c_get_child`

This function returns a child nodes' name and node type (data segment or directory) at the specified offset from the specified path in the storage hierarchy. This function provides a mechanism for "listing the contents" of a directory node.

9.1 Prototype

```
unsigned int sh_c_get child(
        const char * path
        const unsigned int node
        const unsigned int child_offset
        unsigned int * name
        char * type);
```

9.2 Inputs

- path
  The absolute path in the storage hierarchy to the parent of the directory node to be enumerated. The string is null-terminated and cannot exceed MAX_PATH_LENGTH bytes in length.
- node
  The name of the directory node.
- offset
  The offset into the list of children of the input parent node.

9.3 Outputs

- name
  The name of the node at the specified offset as an unsigned int value.
- type
  The type of the node at the specified offset, where NODE_TYPE_DSEG denotes a data segment node and NODE_TYPE_DIRT denotes a directory node.
- Function Result
  A numerical value that indicates success or failure of the function.

9.4 Processing

1. Validate the input path as a correctly formed path string. Return SH_C_MALFORMED_PATH error if validation fails. The path shall only consist of the following characters: /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, NULL.
2. Attempt to walk the storage hierarchy along the specified input path. Return SH_C_PATH_ERROR error if the path specified by the input path does not exist in the storage hierarchy.

3. Verify that a data segment record exists for the input node and path. Return SH_C_NO_SUCH_NODE if it does not
4. Verify that the input path and node specifies a directory node. If it is a data segment node, return SH_C_NODE_NOT_DNODE error.
5. Verify that input offset is not out of bounds of the list of children. Return SH_C_OFFSET_BOUND_ERROR if it is out of bounds.
6. Return the child node name and type.
7. Return NO_ERROR.

9.5 Effects

- This function has no effects.

9.6 Errors

SH_C_MALFORMED_PATH

This error is returned if the input path is not a correctly formed concatenated string of integer values separated by forward-slashes, "/."

SH_C_PATH_ERROR

This error is returned if the input path does not exist in the storage hierarchy tree structure.

SH_C_NO_SUCH_NODE

This error is returned if the combination of input path and node does not exist.

SH_C_NODE_NOT_DNODE

This error is returned if the node specified by the input path is not designated as a directory node.

SH_C_OFFSET_BOUND_ERROR

This error is returned if the specified offset is not within the list of children.

10. sh_c_get_dseg_size

This function returns the length of the specified data segment in bytes.

10.1 Prototype

```
unsigned int sh_c_get_dseg_size(
        const char * const path
        const unsigned int node
        const unsigned int * size);
```

10.2 Inputs

- path
  The absolute path in the storage hierarchy to new child node. The string is null-terminated and cannot exceed MAX_PATH_LENGTH bytes in length.
- node
  The name of the node to retrieve the size for.

10.3 Outputs

- Size
  The size of the data segment in bytes.
- Function Result
  A numerical value that indicates success or failure of the function.

10.4 Processing

1. Validate the input path as a correctly formed path string. Return SH_C_MALFORMED_PATH error if validation fails. The path shall only consist of the following characters: /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, NULL.
2. Attempt to walk the storage hierarchy along the specified input path. Return SH_C_PATH_ERROR error if the path specified by the input path does not exist in the storage hierarchy.
3. Verify that a data segment record exists for the input node and path. Return SH_C_NO_SUCH_NODE if it does not
4. Verify that the specified child node is a data segment node. If it is a directory node, return SH_C_NODE_NOT_DSEG error.
5. Consult the directory node table of the child node's parent node and return the size in bytes for the data segment it reports.
6. Return NO_ERROR.

10.5 Effects

- This function has no effects.

10.6 Errors

SH_C_MALFORMED_PATH

This error is returned if the input path is not a correctly formed concatenated string of integer values separated by forward-slashes, "/."

SH_C_PATH_ERROR

This error is returned if the input path does not exist in the storage hierarchy tree structure.

SH_C_NO_SUCH_NODE

This error is returned if the combination of input path and node does not exist.

SH_C_NODE_NOT_DSEG

This error is returned if the node specified by the input path is not designated as a data segment node.

11. `sh_c_get_dseg_hash`

This function returns the hash value of the specified data segment.

11.1 Prototype

```
unsigned int sh_c_get_dseg_hash(
        const char * const const path
        const unsigned int node
        const void * const buffer);
```

11.2 Inputs

- path
  The absolute path in the storage hierarchy to the data segment node. The string is null-terminated and cannot exceed MAX_PATH_LENGTH bytes in length.
- node
  The name of the node to retrieve the stored hash for.

11.3 Outputs

- buffer
  The 256-bit buffer where the hash value of the data segment will be placed.
- Function Result
  A numerical value that indicates success or failure of the function.

11.4 Processing

1. Verify that the output buffer pointer is not null. Return SH_C_BUFF_PTR_NULL if it is.
2. Validate the input path as a correctly formed path string. Return SH_C_MALFORMED_PATH error if validation fails. The path shall only consist of the following characters: /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, NULL.
3. Attempt to walk the storage hierarchy along the specified input path. Return SH_C_PATH_ERROR error if the path specified by the input path does not exist in the storage hierarchy.
4. Verify that a data segment record exists for the input node and path. Return SH_C_NO_SUCH_NODE if it does not
5. Verify that the specified child node is a data segment node. If it is a directory node, return SH_C_NODE_NOT_DSEG error.
6. Return the hash value stored in meta-data associated with the specified data segment.
7. Return NO_ERROR.

11.5 Effects

- If the output buffer is an invalid pointer, then the behavior of the program is undefined, and will probably lead to termination of the program by the operating system.

11.6 Errors

SH_C_BUFF_PTR_NULL
: This error is returned if the input buffer is NULL.

SH_C_MALFORMED_PATH
: This error is returned if the input path is not a correctly formed concatenated string of integer values separated by forward-slashes, "/."

SH_C_PATH_ERROR
: This error is returned if the input path does not exist in the storage hierarchy tree structure.

SH_C_NO_SUCH_NODE
: This error is returned if the combination of input path and node does not exist.

SH_C_NODE_NOT_DSEG
: This error is returned if the node specified by the input path is not designated as a data segment node.

12. `sh_c_read_dseg`

This function reads the specified number of bytes starting at the offset provided from the data segment associated with the specified handle and writes it into the specified buffer.

12.1 Prototype

```
unsigned int sh_c_read_dseg(
        const unsigned int handle
        const void * const buffer
        const unsigned int byte_count
        const unsigned int offset
        unsigned int * bytes_read);
```

12.2 Inputs

- handle
  The handle to a data segment to be read.
- byte_count
  The number of bytes to read from the data segment.
- offset
  The offset into the data segment to begin reading data.

12.3 Outputs

- bytes_read
  The number of bytes actually read from the specified data segment and placed into the input buffer.
- buffer
  The pointer to the buffer to which data will be written.
- Function Result
  A numerical value that indicates success or failure of the function.

12.4 Processing

1. Set the output bytes_read to zero.
2. Validate the input handle is associated with an open segment. Return SH_C_HANDLE_INVALID if it is not
3. Validate that the buffer is not null. Return SH_C_OUTPUT_BUFF_NULL if the pointer is null.
4. Verify that the specified offset is within the bounds of the specified data segment. Return SH_C_OFFSET_BOUND_ERROR error if it is not.
5. Read the contents of the specified data segment at the specified offset, writing into the specified buffer.
6. Return the number of bytes read from the data segment.
7. Return NO_ERROR.

12.5 Effects

- If the input buffer is an invalid pointer, then the behavior of the program is undefined, and will probably lead to termination of the program by the operating system.

12.6 Errors

SH_C_HANDLE_INVALD

This error if the specified handle is not a valid handle because it does not exist in the open data segment database

SH_C_OFFSET_BOUND_ERROR

This error is returned if the specified input offset is outside the bounds of the data segment.

SH_C_ OUTPUT_BUFF_NULL

This error is returned if the specified buffer cannot be written to.

13. `sh_c_check_hash`

This function calculates the hash of the contents of the specified data segment and compares it with the hash stored in meta-data.

13.1 Prototype

```
unsigned int sh_c_check_hash(
        const char * const path
        const unsigned int node
        unsigned int * result);
```

13.2 Inputs

- path
  The absolute path in the storage hierarchy to a data segment node. The string is null-terminated and cannot exceed MAX_PATH_LENGTH bytes in length.
- node
  The name of the data segment node.

13.3 Outputs

- result
  A value indicating the result of the hash comparison.
- Function Result
  A numerical value that indicates success or failure of the function.

13.4 Processing

1. Validate the input path as a correctly formed path string. Return SH_C_MALFORMED_PATH error if validation fails. The path shall only consist of the following characters: /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, NULL.
2. Attempt to walk the storage hierarchy along the specified input path. Return SH_C_PATH_ERROR error if the path specified by the input path does not exist in the storage hierarchy.
3. Verify that a data segment record exists for the input node and path. Return SH_C_NO_SUCH_NODE if it does not
4. Verify that the specified child node is a data segment node. If it is a directory node, return SH_C_NODE_NOT_DSEG error.
5. Calculate the hash of the contents of the data segment and compare the result with the hash stored in the meta-data associated with the data segment.
6. Set the output result to HASH_EQUAL if the hash calculated is the same as the hash stored, or HASH_NOT_EQUAL if the hashes are not equal.
7. Return NO_ERROR.

13.5 Effects

- This function has no effects.

13.6 Errors

SH_C_MALFORMED_PATH

This error is returned if the input path is not a correctly formed concatenated string of integer values separated by forward-slashes, "/."

SH_C_PATH_ERROR

This error is returned if the input path does not exist in the storage hierarchy tree structure.

SH_C_NO_SUCH_NODE

This error is returned if the input path does not exist in the storage hierarchy tree structure.

SH_C_NODE_NOT_DSEG

This error is returned if the node specified by the input path is not designated as a data segment node.

B. INITIALIZATION INTERFACE

This section describes the functions used by the LPSK during system start up and initialization. The functions are exposed to the LPSK as a series of function calls. They are:

- sh_i_initialize_databases
- sh_i_get_handle
- sh_i_read_in
- sh_i_get_dseg_size
- sh_i_get_dseg_hash

The following subsection details these functions.

1. `sh_i_initialize_databases`

This function reads the storage hierarchy configuration databases from disk in memory.

1.1 Prototype

```
unsigned int sh_i_initialize_databases (
     const unsigned int offset
     const unsigned int num_dsegs);
```

1.2 Inputs

- offset
  The disk block offset into the storage medium where the storage hierarchy configuration data is located.
- num_dsegs
  The number of handles that will be associated with data segments.

1.3 Outputs

- Function Result
  A numerical value that indicates success or failure of the function.

1.4 Processing

1. Attempt to read the configuration database from disk at input offset into memory. If a configuration database was not found, return SH_C_CONFIG_DB_LOAD_ERR error. If there is not enough primary memory, cause the kernel to halt.
2. Attempt to read the SAT from disk into memory. If a SAT structure was not found, return SH_C_SAT_LOAD_ERR. If there is not enough primary memory, cause the kernel to halt.
3. Construct the handle to data segment table in memory with as many rows as input num_dsegs. If there is not enough primary memory, cause the kernel to halt.
4. Return NO_ERROR.

1.5 Effects

- The amount of available memory will be reduced by the amount needed for configuration database, SAT, and handle table.
- The platform may halt if insufficient memory is available.

1.6 Errors

SH_C_CONFIG_DB_LOAD_ERR

This error is returned if the data loaded from disk does not represent a configuration database structure.

SH_C_CLUS_DB_LOAD_ERR

This error is returned if the data loaded from disk does not represent a cluster database structure.

SH_C_SAT_LOAD_ERR

This error is returned if the data loaded from disk does not represent a SAT structure.

2. `sh_i_get_handle`

This function associates a specified path to a data segment with a new handle, which is returned to the caller.

2.1 Prototype

```
unsigned int sh_i_get_handle(
    const char * const path
    const unsigned int node
    const unsigned int * handle);
```

2.2 Inputs

- path
  The absolute path in the storage hierarchy to the parent of the data segment to associate with a new handle. The string is null-terminated and cannot exceed MAX_PATH_LENGTH bytes in length.
- node
  The name of the node to associate with a handle.

2.3 Outputs

- handle
  The handle associated with the specified path.
- Function Result
  A numerical value that indicates success or failure of the function.

2.4 Processing

1. Validate that the handle table has been initialized. Cause the kernel to halt if the handle table has not been initialized.
2. Validate that the specified path has not already been associated with a handle. Cause the kernel to halt if it has.
3. Cause the kernel to halt if there are no handles available.
4. Validate the input path as a correctly formed path string. Return SH_I_MALFORMED_PATH error if validation fails.
5. Attempt to walk the storage hierarchy along the specified input path. Return SH_I_PATH_ERROR error if the path specified by the input path does not exist in the storage hierarchy.
6. Verify that a data segment record exists for the input node and path. Return SH_C_NO_SUCH_NODE if it does not
7. Verify that the specified child node is a data segment node. If it is a directory node, return SH_I_NODE_NOT_DSEG error.
8. Associate an entry in the handle to path database and populate it with the input path/node and the data segment meta-data.
9. Set the output handle with a handle to the entry in the handle to path database.
10. Return NO_ERROR

2.5 Effects

- An association is made between a data segment and handle in the handle-to-path data database and data segment meta-data is loaded in memory.
- The platform may potentially halt if this interface is called before the handle table has been initialized or if multiple handles are requested for the same data segment .

2.6 Errors

SH_I_MALFORMED_PATH
: This error is returned if the input path is not a correctly formed concatenated string of integer values separated by forward-slashes, "/."

SH_I_PATH_ERROR
: This error is returned if the input path does not exist in the storage hierarchy tree structure.

SH_C_NO_SUCH_NODE
: This error is returned if the combination of input path and node does not exist.

SH_I_NODE_NOT_DSEG
: This error is returned if the node specified by the input path is not designated as a data segment node.

3. `sh_i_read_in`

This function supports the swap in functional requirement by copying the data segment from secondary storage associated with the specified handle into the specified buffer.

3.1 Prototype

```
unsigned int sh_i_read_in(
    const unsigned int handle
    const void * buffer);
```

3.2 Inputs

- handle
  The handle associated with the desired data segment in the storage hierarchy.
- buffer
  The memory buffer the data segment will be copied into.

3.3 Outputs

- Function Result
  A numerical value that indicates success or failure of the function.

3.4 Processing

1. Validate that the handle table has been initialized. Cause the kernel to halt if it has not been initialized.
2. Validate the input handle is a member of the handle-to-path data structure. Return SH_R_HANDLE_INVALID error if it is not a member.
3. Read data segment from areas of secondary storage specified by the meta-data associated with the handle.
4. Copy the data associated with the specified handle into the output buffer.
5. Return NO_ERROR.

3.5 Effects

- The contents of a segment in primary memory is modified.
- The platform may potentially halt if this interface is called before the handle table has been initialized.

3.6 Errors

SH_R_HANDLE_INVALID

This error is returned if the value of input handle does not exist in the handle-to-path data structure of the storage hierarchy.

4. `sh_i_get_dseg_size`

This function returns the size of the specified data segment.

4.1 Prototype

```
unsigned int sh_r_get_dseg_size(
        const unsigned int handle
        const unsigned int * size);
```

4.2 Inputs

- handle
  The handle associated with the desired data segment in the storage hierarchy.

4.3 Outputs

- size
  The size of the specified data segment in bytes.
- Function Result
  A numerical value that indicates success or failure of the function.

4.4 Processing

1. Validate that the handle table has been initialized. Cause the kernel to halt if it has not been initialized.
2. Validate the input handle is a member of the handle-to-path data structure. Return SH_R_HANDLE_INVALID error if it is not a member.
3. Retrieve the path associated with the handle in the handle-to-path database.
4. Using the input handle to reference the handle to path table, set the output size with the size cached in the handle to path table.
5. Return NO_ERROR.

4.5 Effects

The platform may potentially halt if this interface is called before the handle table has been initialized.

4.6 Errors

SH_R_HANDLE_INVALID

This error is returned if the value of input handle does not exist in the handle-to-path data structure of the storage hierarchy.

5. `sh_i_get_dseg_hash`

This function returns the hash value of the specified data segment.

5.1 Prototype

```
unsigned int sh_i_get_dseg_hash(
        const unsigned int handle
        const void * const buffer);
```

5.2 Inputs

- handle
  The handle associated with the desired data segment in the storage hierarchy.
- buffer
  The 256-bit buffer where the hash value of the data segment will be placed.

5.3 Outputs

- Function Result
  A numerical value that indicates success or failure of the function.

5.4 Processing

1. Validate that the handle table has been initialized. Cause the kernel to halt if it has not been initialized.
2. Validate the input handle is a member of the handle-to-path data structure. Return SH_R_HANDLE_INVALID error if it is not a member.
3. Using the input handle to reference the handle to path table, set the output buffer with the hash value cached in the handle to table database.
4. Return NO_ERROR.

5.5 Effects

- The platform may potentially halt if this interface is called before the handle table has been initialized..

5.6 Errors

SH_R_HANDLE_INVALID

This error is returned if the value of input handle does not exist in the handle-to-path data structure of the storage hierarchy.

C. RUNTIME INTERFACE

This section describes the functions used by the LPSK during run-time. The functions are exposed to the LPSK as a series of function calls. They are:

- sh_r_read_in
- sh_r_write_out
- sh_r_get_dseg_size
- sh_r_get_dseg_hash
- sh_r_check_hash

The following subsections detail these functions.

1. `sh_r_read_in`

This function supports the swap in functional requirement by copying the data segment from secondary storage associated with the specified handle into the specified buffer.

1.1 Prototype

```
unsigned int sh_r_read_in(
    const unsigned int handle
    const void * buffer);
```

1.2 Inputs

- handle
  The handle associated with the desired data segment in the storage hierarchy.
- buffer
  The memory buffer the data segment will be copied into.

1.3 Outputs

- Function Result
  A numerical value that indicates success or failure of the function.

1.4 Processing

1. Validate that the handle table has been initialized. Cause the kernel to halt if it has not been initialized.
2. Validate the input handle is a member of the handle-to-path data structure. Return SH_R_HANDLE_INVALID error if it is not a member.
3. Read data segment from areas of secondary storage specified by the meta-data associated with the handle.
4. Copy the data segment associated with the specified handle into the output buffer.
5. Return NO_ERROR.

1.5 Effects

- The contents of a segment in primary memory are modified.
- The platform may potentially halt if this interface is called before the handle table has been initialized.

1.6 Errors

SH_R_HANDLE_INVALID

This error is returned if the value of input handle does not exist in the handle-to-path data structure of the storage hierarchy.

2. `sh_r_write_out`

This function satisfies the functional requirements of both swap out and flush calls by copying the contents of the specified buffer into the data segment associated with the specified handle and calculating a new hash value.

2.1 Prototype

```
unsigned int sh_r_write_out(
    const unsigned int handle
    const void * const buffer);
```

2.2 Inputs

- handle
  The handle associated with the desired data segment in the storage hierarchy.
- buffer
  The memory buffer the data segment will be copied from.

2.3 Outputs

- Function Result
  A numerical value that indicates success or failure of the function.

2.4 Processing

1. Validate that the handle table has been initialized. Cause the kernel to halt if it has not been initialized.
2. Validate the input handle is a member of the handle-to-path data structure. Return SH_R_HANDLE_INVALID error if it is not a member.
3. Retrieve secondary storage location information for data segment associated with the specified handle.
4. Copy the contents of the specified buffer into secondary storage.
5. Calculate and store the hash value of the contents of the data segment into the associated meta-data field, return NO_ERROR.

2.5 Effects

- Portions of the secondary storage will be overwritten with the contents of the input buffer.
- The hash value associated with the data segment is updated after new data has been written.
- The platform may potentially halt if this interface is called before the handle table has been initialized

2.6 Errors

SH_R_HANDLE_INVALID

This error is returned if the value of input handle does not exist in the handle-to-path data structure of the storage hierarchy.

3. `sh_r_get_dseg_size`

This function returns the size of the specified data segment.

3.1 Prototype

```
unsigned int sh_r_get_dseg_size(
        const unsigned int handle
        const unsigned int * size);
```

3.2 Inputs

- handle
  The handle associated with the desired data segment in the storage hierarchy.

3.3 Outputs

- size
  The size of the specified data segment in bytes.
- Function Result
  A numerical value that indicates success or failure of the function.

3.4 Processing

1. Validate that the handle table has been initialized. Cause the kernel to halt if it has not been initialized.
2. Validate the input handle is a member of the handle-to-path data structure. Return SH_R_HANDLE_INVALID error if it is not a member.
3. Retrieve the path associated with the handle in the handle-to-path database.
4. Using the input handle to reference the handle to path table, set the output size with the size cached in the handle to path table.
5. Return NO_ERROR.

3.5 Effects

- The platform may potentially halt if this interface is called before the handle table has been initialized.

3.6 Errors

SH_R_HANDLE_INVALID

This error is returned if the value of input handle does not exist in the handle-to-path data structure of the storage hierarchy.

4. `sh_r_get_dseg_hash`

This function returns the hash value of the specified data segment.

4.1 Prototype

```
unsigned int sh_r_get_dseg_hash(
        const unsigned int handle
        const void * const buffer);
```

4.2 Inputs

- handle
  The handle associated with the desired data segment in the storage hierarchy.
- buffer
  The 256-bit buffer where the hash value of the data segment will be placed.

4.3 Outputs

- Function Result
  A numerical value that indicates success or failure of the function.

4.4 Processing

1. Validate that the handle table has been initialized. Cause the kernel to halt if it has not been initialized.
2. Validate the input handle is a member of the handle-to-path data structure. Return SH_R_HANDLE_INVALID error if it is not a member.
3. Using the input handle to reference the handle to path table, set the output buffer with the hash value cached in the handle to table database.
4. Return NO_ERROR.

4.5 Effects

- The platform may potentially halt if this interface is called before the handle table has been initialized.

4.6 Errors

SH_R_HANDLE_INVALID

This error is returned if the value of input handle does not exist in the handle-to-path data structure of the storage hierarchy.

5. `sh_r_check_hash`

This function calculates the hash of the contents of the data segment associated with the specified handle and compares it with the hash stored in meta-data.

5.1 Prototype

```
unsigned int sh_r_check_hash(
        const unsigned int handle
        unsigned int * result);
```

5.2 Inputs

- handle
  The handle of the data segment.

5.3 Outputs

- result
  A value indicating the result of the hash comparison.
- Function Result
  A numerical value that indicates success or failure of the function.

5.4 Processing

1. Validate that the handle table has been initialized. Cause the kernel to halt if it has not been initialized.
2. Validate the input handle is a member of the handle-to-path data structure. Return SH_R_HANDLE_INVALID error if it is not a member.
3. Calculate the hash of the contents of the data segment and compare the result with the hash stored in the meta-data associated with the data segment.
4. Set the output result to HASH_EQUAL if the hash calculated is the same as the hash stored, or HASH_NOT_EQUAL if the hashes are not equal.
5. Return NO_ERROR.

5.5 Effects

- The platform may potentially halt if this interface is called before the handle table has been initialized.

5.6 Errors

SH_R_HANDLE_INVALID

This error is returned if the value of input handle does not exist in the handle-to-path data structure of the storage hierarchy.

APPENDIX B: TEST PROCEDURES

This appendix contains the test procedures used to produce the results located in Chapter IV of this work.

A. TEST PROCEDURES

When an interface is invoked as part of a test procedure, it is described in this appendix as a function call accompanied by one or more parameters. Some test procedures may instruct the reader to perform specific steps from a previous procedure in the case of large amounts of redundancy in procedures. Each test procedure indicates the associated test program argument(s), the values of which are found in the header files of the configuration and LPSK test programs.

1. Configuration Interfaces

These test procedures are performed using the test program compiled with configuration interfaces and modules. Test procedures indicate which test program they use to perform which steps of the procedure

`sh_c_initialize_databases` interface test procedure: These tests correspond to Test Group 1 as described in Table 14.

Begin

Configuration test program (argument value: `TEST_C1`)

1. Call `format` function.
2. Call `sh_c_initialize_databases` function with `block_offset = 8`.
3. Confirm no error returned (test C1A).
4. Call `format` function.
5. Call `sh_c_initialize_databases` function with `block_offset = 0`.
6. Confirm error `SH_C_CONFIG_DB_LOAD_ERR` returned (test C1B).
7. Call `format` function with `cluster_database_offset = 0`.

8. Call `sh_c_initialize_databases` function with `block_offset = 8`.
9. Confirm error `SH_C_CLUS_DB_LOAD_ERR` returned (test C1C).
10. Call `format` function with `SAT_offset = 0`.
11. Call `sh_c_initialize_databases` function with `block_offset = 8`.
12. Confirm error `SH_C_SAT_LOAD_ERR` returned (test C1D).

End

`sh_c_make_dseg` interface test procedure: These tests correspond to Test Group 2 as described in Table 15.

Begin

Configuration test program (argument value: `TEST_C2`)

1. Call `format` function.
2. Call `sh_c_initialize_databases` function with `block_offset = 8`.
3. Call `sh_c_make_dseg` function with `path = "/"`, `node = 0`, and `size = 9000`.
4. Confirm no error returned (test C2A.1).
5. Call `sh_c_make_dseg` function with `path = "/"`, `node = 0`, and `size = 9000`.
6. Confirm error `SH_C_NODE_ALREADY_EXISTS` error returned (test C2A.2).
7. Call format function
8. Call `sh_c_initialize_databases` function with `block_offset = 8`.
9. Call `sh_c_make_dseg` function with `path = "/"`, `node = 128`, and `size = 9,000`.
10. Confirm error `_SH_C_MAX_NODE_NAME` returned (test C2B).
11. Call `sh_c_make_dseg` function with `path = "/"`, `node = 0`, and `size = 134217728`.
12. Confirm error `SH_C_DISKSPACE` returned (test C2C).
13. Call `sh_c_make_dseg` function with `path = "a"`, `node = 0`, and `size = 9000`.

14. Confirm error `SH_C_MALFORMED_PATH` returned (test C2D).
15. Call `sh_c_make_dseg` function with `path = “/a”`, `node = 0`, and `size = 9000`.
16. Confirm error `SH_C_MALFORMED_PATH` returned (test C2E).
17. Call `sh_c_make_dseg` function with `path = “/0”`, `node = 0`, and `size = 9000`.
18. Confirm error `SH_C_PATH_ERROR` returned (test C2F).

<u>End</u>

`sh_c_make_subtree` interface test procedure: These tests correspond to Test Group 3 as described in Table 16.

<u>Begin</u>

Configuration test program (argument value: `TEST_C3`)

1. Call format function.
2. Call `sh_c_initialize_databases` function with `block_offset = 8`.
3. Call `sh_c_make_subtree` function with `path = “/”` and `node = 0`.
4. Confirm no error returned (test C3A.1).
5. Call `sh_c_make_subtree` function with `path = “/”` and `node = 0`.
6. Confirm error `SH_C_NODE_ALREADY_EXISTS` returned (test C3A.2).
7. Call `sh_c_make_subtree` function with `path = “/”` and `node = 128`.
8. Confirm error `_SH_C_MAX_NODE_NAME` returned (test C3B).
9. Call format function
10. Call `sh_c_initialize_databases` function with `block_offset = 8`.
11. Call `sh_c_make_dseg` function with `path = “/”`, `node = 1`, and `size = 134209536`.
12. Call `sh_c_make_subtree` function with `path = “/”` and `node = 0`.
13. Confirm error `SH_C_DISKSPACE` returned (test C3C).

14. Call `sh_c_make_subtree` function with `path = “a”`.
15. Confirm error `SH_C_MALFORMED_PATH` returned (test C3D).
16. Call `sh_c_make_subtree` function with `path = “/a”`.
17. Confirm error `SH_C_MALFORMED_PATH` returned (test C3E).
18. Call `sh_c_make_subtree` function with `path = “/0”`.
19. Confirm error `SH_C_PATH_ERROR` returned (test C3F).

End

`sh_c_open_dseg` interface test procedure: These tests correspond to Test Group 4 as described in Table 17.

Begin

Configuration test program (argument value: `TEST_C4`)

1. Call format function.
2. Set constant `MAX_NUM_H_ROWS` in configuration interfaces module to 2.
3. Call `sh_c_initialize_databases` function with `block_offset = 8`.
4. Call `sh_c_make_dseg` function with `path = “/”` and `node = 1`.
5. Call `sh_c_make_dseg` function with `path = “/”` and `node = 2`.
6. Call `sh_c_make_dseg` function with `path = “/”` and `node = 3`.
7. Call `sh_c_open_dseg` function with `path = “/”` and `node = 1`.
8. Confirm no error returned; handle zero returned (test C4A.1).
9. Call `sh_c_open_dseg` function with `path = “/”` and `node = 2`.
10. Confirm no error returned; handle one returned (test C4A.2).
11. Call `sh_c_open_dseg` function with `path = “/”` and `node = 3`.
12. Confirm error `SH_C_HANDLE_TABLE_FULL` returned (test C4A.3).
13. Call format function.
14. Call `sh_c_initialize_database` function with `block_offset = 8`.
15. Call `sh_c_open_dseg` function with `path = “/”` and `node = 10`.
16. Confirm error `SH_C_NO_SUCH_NODE` returned (test C4B).
17. Call `sh_c_make_subtree` function with `path = “a”`.
18. Confirm error `SH_C_MALFORMED_PATH` returned (test C4C).

19. Call `sh_c_make_subtree` function with `path = "/a"`.
20. Confirm error `SH_C_MALFORMED_PATH` returned (test C4D).
21. Call `sh_c_make_subtree` function with `path = "/0"`.
22. Confirm error `SH_C_PATH_ERROR` returned (test C4E).

End

`sh_c_write_dseg` interface test procedure: These tests correspond to Test Group 5 as described in Table 18.

Begin

Configuration test program (argument value: `TEST_C5`)

1. Call format function.
2. Call `sh_c_initialize_databases` function with `block_offset = 8`.
3. Call `sh_c_make_dseg` function with `path = "/"`, `node = 1`, and `size = 9000`.
4. Call `sh_c_open_dseg` function with `path = "/"` and `node = 1`; handle zero returned.
5. Create first 9,000 byte buffer in memory and fill with 9,000 *a* characters; assign pointer `buf_ptr1` to buffer.
6. Generate and store hash of first buffer.
7. Call `sh_c_make_dseg` function with `path = "/"`, `node = 2`, and `size = 9000`.
8. Call `sh_c_open_dseg` function with `path = "/"` and `node = 2`; handle one returned.
9. Create second 9,000 byte buffer in memory and fill with 9,000 *b* characters; assign pointer `buf_ptr2` to buffer.
10. Generate and store hash of second buffer.
11. Call `sh_c_write_dseg` function with `handle = 0`, `buffer = buf_ptr1`, `byte_count = 9000`, and `offset = 0`.
12. Confirm no error returned (test C5A).
13. Call `sh_c_write_dseg` function with `handle = 1`, `buffer = buf_ptr2`, `byte_count = 4500`, and `offset = 0`.
14. Confirm no error returned (test C5B.1).

15. Call `sh_c_write_dseg` function with `handle = 1`, `buffer = buf_ptr2`, `byte_count = 4500`, and `offset = 4500`.
16. Confirm no error returned (test C5B.2).
17. Call `sh_c_write__data` function segment with `handle = 2`, `buffer = buf_ptr1`, `byte_count = 9000`, and `offset = 0`.
18. Confirm error `SH_C_HANDLE_INVALID` returned (test C5C).
19. Call `sh_c_write_dseg` function with `handle = 0`, `buffer = buf_ptr1`, `byte_count = 9000`, and `offset = 9200`.
20. Confirm error `SH_C_OFFSET_BOUND_ERROR` returned (test C5D).
21. Call `sh_c_write_dseg` function with `handle = 0`, `buffer = buf_ptr1`, `byte_count = 8000`, and `offset = 2000`.
22. Confirm error `SH_C_INSUFFICENT_SPACE` returned (test C5E).
23. Call `sh_c_write_dseg` function with `handle = 0`, `buffer = NULL`, `byte_count = 0`, and `offset = 0`.
24. Confirm error `SH_C_INPUT_BUFF_NULL` returned (test C5F).

End

`sh_c_read_dseg` interface test procedure: These tests correspond to Test Group 6 as described in Table 19.

Begin

Configuration test program (argument value: `TEST_C6`)

1. Repeat steps 1 through 16 from `sh_c_write_dseg` test procedure.
2. Call `sh_c_read_dseg` function with `handle = 0`, `buffer = buf_ptr1`, `byte_count = 9000`, and `offset = 0`.
3. Confirm no error returned; generate new hash of first buffer and confirm it matches previously generated hash to determine if (test C6A).
4. Call `sh_c_read_dseg` function with `handle = 1`, `buffer = buf_ptr2`, `byte_count = 4500`, and `offset = 0`.
5. Confirm no error returned (test C6B.1).
6. Call `sh_c_read_dseg` function with `handle = 1`, `buffer = buf_ptr2`, `byte_count = 4500`, and `offset = 4500`.
7. Confirm no error returned; generate new hash of second buffer and confirm it matches previously generated hash (test C6B.2).

8. Call `sh_c_read_dseg` function with `handle = 2`, `buffer = buf_ptr1`, `byte_count = 9000`, and `offset = 0`.
9. Confirm error `SH_C_HANDLE_INVALID` returned (test C6C).
10. Call `sh_c_read_dseg` function with `handle = 0`, `buffer = buf_ptr1`, `byte_count = 9000`, and `offset = 9200`.
11. Confirm error `SH_C_OFFSET_BOUND_ERROR` returned (test C6D).
12. Call `sh_c_read_dseg` function with `handle = 0`, `buffer = NULL`, `byte_count = 0`, and `offset = 0`.
13. Confirm error `SH_C_OUTPUT_BUFF_NULL` returned (test C6E).

End

`sh_c_close_dseg` interface test procedure: These tests correspond to Test Group 7 as described in Table 20.

Begin

Configuration test program (argument value: `TEST_C7`)

1. Repeat steps 1 through 6 from `sh_c_write_dseg` test procedure.
2. Call `sh_c_close_dseg` function with `handle = 0`.
3. Confirm no error returned.
4. Call `sh_c_get_dseg_hash` function with `path = "/"` and `node = 1`; compare returned hash with previous generated hash (test C7A.1).
5. Call `sh_c_close_dseg` function with `handle = 1`.
6. Confirm error `SH_C_HANDLE_INVALID` returned (test C7A.2).
7. Call `sh_c_close_dseg` function with `handle = 2`.
8. Confirm error `SH_C_HANDLE_INVALID` returned (test C7B).

End

`sh_c_delete_dseg` interface test procedure: These tests correspond to Test Group 8 as described in Table 21.

Begin

Configuration test program (argument value: `TEST_C8`)

1. Call format function.
2. Call `sh_c_initialize_databases` function with `block_offset = 8`.

3. Call `sh_c_make_dseg` function with `path = "/"`, `node = 1`, and `size = 9000`.
4. Call `sh_c_make_subtree` function with `path = "/"` and `node = 2`.
5. Call `sh_c_delete_dseg` function with `path = "/"` and `node = 1`.
6. Confirm no error returned (test C8A.1).
7. Call `sh_c_delete_dseg` function with `path = "/"` and `node = 1`.
8. Confirm error `SH_C_NO_SUCH_NODE` returned (test C8A.2).
9. Call `sh_c_delete_dseg` function with `path = "/"` and `node = 128`.
10. Confirm error `SH_C_MAX_NODE_NAME` returned (test C8B).
11. Call `sh_c_delete_dseg` function with `path = "/"` and `node = 0`.
12. Confirm error `SH_C_NO_SUCH_NODE` returned (test C8C).
13. Call `sh_c_delete_dseg` function with `path = "/"` and `node = 2`.
14. Confirm error `SH_C_NODE_NOT_DSEG` returned (test C8D).
15. Call `sh_c_delete_dseg` function with `path = "a"`.
16. Confirm error `SH_C_MALFORMED_PATH` returned (test C8E).
17. Call `sh_c_delete_dseg` function with `path = "/a"`.
18. Confirm error `SH_C_MALFORMED_PATH` returned (test C8F).
19. Call `sh_c_delete_dseg` function with `path = "/0"`.
20. Confirm error `SH_C_PATH_ERROR` returned (test C8G).

End

`sh_c_delete_subtree` interface test procedure: These tests correspond to Test Group 9 as described in Table 22.

Begin

Configuration test program (argument value: `TEST_C9`)

1. Call format function.

2. Call `sh_c_initialize_databases` function with `block_offset = 8`.
3. Call `sh_c_make_subtree` function with `path = "/"` and `node = 1`.
4. Call `sh_c_make_subtree` function with `path = "/1"` and `node = 1`.
5. Call `sh_c_make_subtree` function with `path = "/1"` and `node = 2`.
6. Call `sh_c_make_dseg` function with `path = "/1/1"` and `node = 1`.
7. Call `sh_c_make_dseg` function with `path = "/1/2"` and `node = 1`.
8. Call `sh_c_delete_subtree` function with `path = "/1/1"` and `node = 1`.
9. Confirm error `SH_C_NODE_DIRNODE` returned (test C9A).
10. Call `sh_c_delete_subtree` function with `path = "/"` and `node = 1.`
11. Confirm no error returned (test C9B.1).
12. Call sh_c_delete subtree with `path = "/"` and `node = 1`.
13. Confirm error `SH_C_NO_SUCH_NODE` returned (test C9B.2).
14. Call sh_c_delete subtree with `path = "/1"` and `node = 1`.
15. Confirm error `SH_C_PATH_ERROR` returned (test C9B.3).
16. Call sh_c_delete subtree with `path = "/"` and `node = 128`.
17. Confirm error `SH_C_MAX_NODE_NAME` returned (test C9C).
18. Call sh_c_delete subtree with `path = "/"` and `node = 0`.
19. Confirm error `SH_C_NO_SUCH_NODE` returned (test C9D).
20. Call `sh_c_delete_subtree` function with `path = "a"`.
21. Confirm error `SH_C_MALFORMED_PATH` returned (test C9E).
22. Call `sh_c_delete_subtree` function with `path = "/a"`.
23. Confirm error `SH_C_MALFORMED_PATH` returned (test C9F).
24. Call `sh_c_delete_subtree` function with `path = "/0"`.
25. Confirm error `SH_C_PATH_ERROR` returned (test C9G).

End

`sh_c_get_child` interface test procedure: These tests correspond to Test Group 10 as described in Table 23.

Begin

Configuration test program (argument value: `TEST_C10`)

1. Repeat steps 1 through 7 from `sh_c_delete_subtree` test procedure.
2. Call `sh_c_get_child` function with `path = "/"`, `node = 1`, and `offset = 0`.
3. Confirm no error returned; expected node name `1` and directory node type returned (test C10A).
4. Call `sh_c_get_child` function with `path = "/"`, `node = 1`, and `offset = 1`.
5. Confirm no error returned; expected node name `2` and directory node type returned (test C10B).
6. Call `sh_c_get_child` function with `path = "/"`, `node = 1`, and `offset = 2`.
7. Confirm error `SH_C_OFFSET_BOUND_ERROR` returned (test C10C).
8. Call `sh_c_get_child` function with `path = "/1"`, `node = 1`, and `offset = 0`.
9. Confirm no error returned; expected node name 1 and data segment node type returned (C10D).
10. Call `sh_c_get_child` function with `path = "/1/1"`, `node = 1`, and `offset = 0`.
11. Confirm error `SH_C_NODE_NOT_DSEG` returned (test C10E).
12. Call `sh_c_get_child` function with `path = "/"`, `node = 128`, and `offset = 0`.
13. Confirm error `SH_C_MAX_NODE_NAME` returned (test C10F).
14. Call `sh_c_get_child` function with `path = "/"`, `node = 0`, and `offset = 0`.
15. Confirm error `SH_C_NO_SUCH_NODE` returned (test C10G).
16. Call `sh_c_delete_subtree` function with `path = "a"`.
17. Confirm error `SH_C_MALFORMED_PATH` returned (test C10H).
18. Call `sh_c_delete_subtree` function with `path = "/a"`.

19. Confirm error `SH_C_MALFORMED_PATH` returned (test C10I).
20. Call `sh_c_delete_subtree` function with `path = “/0”`.
21. Confirm error `SH_C_PATH_ERROR` returned (test C10J).

End

`sh_c_get_dseg_size` interface procedure test: These tests correspond to Test Group 11 as described in Table 24.

Begin

Configuration test program (argument value: `TEST_C11`)

1. Call format function
2. Call `sh_c_initialize_databases` function with `block_offset = 8`.
3. Call `sh_c_make_dseg` function with `path = “/”`, `node = 1`, and `size = 9000`.
4. Call `sh_c_make_subtree` function with `path = “/”` and `node = 2`.
5. Call `sh_c_get_dseg_size` function with `path = “/”` and `node = 1`.
6. Confirm no error returned; 9,000 bytes returned (test C11A).
7. Call `sh_c_get_dseg_size` function with `path = “/”` and `node = 2`.
8. Confirm error `SH_C_NODE_NOT_DSEG` returned (test C11B).
9. Call `sh_c_get_dseg_size` function with `path = “/”` and `node = 3`.
10. Confirm error `SH_C_NO_SUCH_NODE` returned (test C11C).
11. Call `sh_c_get_dseg_size` function with `path = “/”` and `node = 128`.
12. Confirm error `SH_C_MAX_NODE_NAME` returned (test C11D).
13. Call `sh_c_get_dseg_size` function with `path = “a”`.
14. Confirm error `SH_C_MALFORMED_PATH` returned (test C11E).
15. Call `sh_c_get_dseg_size` function with `path = “/a”`.
16. Confirm error `SH_C_MALFORMED_PATH` returned (test C11F).
17. Call `sh_c_get_dseg_size` function with `path = “/0”`.

18. Confirm error `SH_C_PATH_ERROR` returned (test C11G).

End

`sh_c_get_dseg_hash` interface procedure test: These tests correspond to Test Group 12 as described in Table 25.

Begin

Configuration test program (argument value: `TEST_C12`)

1. Repeat steps 1 through 4 from `sh_c_get_dseg_size` test procedure.
2. Call `sh_c_open_dseg` function with `path = "/"` and `node = 1`; handle zero returned.
3. Create 9,000 byte buffer in memory and fill with 9,000 *a* characters; assign pointer `buf_ptr1` to buffer.
4. Generate and store hash of buffer.
5. Call `sh_c_write_dseg` function with `handle = 0`, `buffer = buf_ptr1`, `byte_count = 9000`, and `offset = 0`.
6. Call `sh_c_close_dseg` function with `handle = 0`.
7. Call `sh_c_make_subtree` function with `path = "/"` and `node = 2`.
8. Call `sh_c_get_dseg_hash` function with `path = "/"` and `node = 1`.
9. Confirm no error returned; compare returned hash with previous generated hash (test C12A).
10. Call `sh_c_get_dseg_hash` function with `path = "/"` and `node = 2`.
11. Confirm error `SH_C_NODE_NOT_DSEG` returned (test C12B).
12. Call `sh_c_get_dseg_hash` function with `path = "/"` and `node = 3`.
13. Confirm error `SH_C_NO_SUCH_NODE` returned (test C12C).
14. Call `sh_c_get_dseg_hash` function with `path = "/"` and `node = 128`.
15. Confirm error `SH_C_MAX_NODE_NAME` returned (test C12D).
16. Call `sh_c_get_dseg_hash` function with `path = "a"`.
17. Confirm error `SH_C_MALFORMED_PATH` returned (test C12E).
18. Call `sh_c_get_dseg_hash` function with `path = "/a"`.

19. Confirm error `SH_C_MALFORMED_PATH` returned (test C12F).
20. Call `sh_c_get_dseg_hash` function with `path = "/0"`.
21. Confirm error `SH_C_PATH_ERROR` returned (test C12G).

End

`sh_c_check_hash` interface test procedure: These tests correspond to Test Group 13 as described in Table 26.

Begin

Configuration test program (argument value: `TEST_C13`)

1. Repeat steps 1 through 7 from `sh_c_get_dseg_hash` test procedure.
2. Call `sh_c_check_hash` function with `path = "/"`, and `node = 1`.
3. Confirm no error returned; confirm true value returned (test C13A)
4. Call `sh_c_check_hash` function with `path = "/"` and `node = 2`.
5. Confirm error `SH_C_NODE_NOT_DSEG` returned (test C13B).
6. Call `sh_c_check_hash` function with `path = "/"` and `node = 3`.
7. Confirm error `SH_C_NO_SUCH_NODE` returned (test C13C).
8. Call `sh_c_check_hash` function with `path = "/"` and `node = 128`.
9. Confirm error `SH_C_MAX_NODE_NAME` returned (test C13D).
10. Call `sh_c_check_hash` function with `path = "a"`.
11. Confirm error `SH_C_MALFORMED_PATH` returned (test C13E).
12. Call `sh_c_check_hash` function with `path = "/a"`.
13. Confirm error `SH_C_MALFORMED_PATH` returned (test C13F).
14. Call `sh_c_check_hash` function with `path = "/0"`.
15. Confirm error `SH_C_PATH_ERROR` returned (test C13G).

End

2. Initialization Interfaces

Initialization interface test procedures require use of both configuration and LPSK test tools described in Chapter IV. Test procedures indicate which test program they use to perform which steps of the procedure.

`sh_i_initialize_databases` interface test procedures: These tests correspond to Test Group 1 as described in Table 27.

<u>Begin</u>

Configuration test program (argument value: `TEST_I1A`):

1. Call format function.

LPSK test program (argument value: `TEST_I1A)`:

2. Call `sh_i_initialize_databases` function with `block_offset = 8`.
3. Confirm no error returned (test I1A).
4. Restart LPSK test program:
5. Call `sh_i_initialize_databases` function with `block_offset = 0` and `num_dsegs = 1`.
6. Confirm error `SH_C_CONFIG_DB_LOAD_ERR` returned (test I1B).

Configuration test program (argument value: `TEST_I1B`):

7. Call format function with `cluster_database_offset = 0`.

LPSK test program (argument value: `TEST_I1B)`:

8. Call `sh_i_initialize_databases` function with `block_offset = 8` and `num_dsegs = 1`.
9. Confirm error `SH_C_CLUS_DB_LOAD_ERR` returned (test I1C).

Configuration test program (argument value: `TEST_I1B`):

10. Call format function with `SAT_offset = 0`.

LPSK test program (argument value: `TEST_I1B)`:

11. Call `sh_i__initialize_databases` function with `block_offset = 8` and `num_dsegs = 1`.
12. Confirm error `SH_C_SAT_LOAD_ERR` returned (test I1D).

<u>End</u>

`sh_i_get_handle` interface test procedure: These tests correspond to Test Group 2 as described in Table 28.

<u>Begin</u>

Configuration test program (argument value: `TEST_I2`):

1. Call format function.
2. Call `sh_c_sh_c_initialize_databases` function with `block_offset = 8`.
3. Call `sh_c_sh_c_make_dseg` function with `path = "/"` and `node = 1`.
4. Call `sh_c_sh_c_make_dseg` function with `path = "/"` and `node = 2`.
5. Call `sh_c_sh_c_make_subtree` function with `path = "/"` and `node = 3`.

LPSK test program (argument value: `TEST_I2`):

6. Call `sh_i_initialize_databases` function with `block_offset = 8` and `num_dsegs = 1`.
7. Call `sh_i_get_handle` function with `path = "/"` and `node = 1`.
8. Confirm no error returned; handle zero returned (test I2A.1).
9. Call `sh_i_get_handle` function with `path = "/"` and `node = 2`.
10. Confirm error `SH_C_HANDLE_TABLE_FULL` returned (test I2A.2).
11. Call `sh_i_initialize_databases` function with `block_offset = 8` and `num_dsegs = 1`.
12. Call `sh_i_get_handle` function with `path = "/"` and `node = 3`.
13. Confirm error `SH_C_NODE_NOT_DSEG` returned (test I2B)
14. Call `sh_i_get_handle` function with `path = "/"` and `node = 10`.
15. Confirm error `SH_C_NO_SUCH_NODE` returned (test I2C).
16. Call `sh_i_get_handle` function with `path = "/"` and `node = 128`.
17. Confirm error `SH_C_MAX_NODE_NAME` returned (test I2D).
18. Confirm error `SH_C_MALFORMED_PATH` returned (test I2E).
19. Call `sh_i_check_hash` function with `path = "/a"`.
20. Confirm error `SH_C_MALFORMED_PATH` returned (test I2F).

21. Call `sh_i_check_hash` function with `path = “/0”`.
22. Confirm error `SH_C_PATH_ERROR` returned (test I2G).

End

`sh_i_read_in` interface test procedure: These tests correspond to Test Group 3 as described in Table 29.

Begin

Configuration test program (argument value: `TEST_I3`):

1. Call format function.
2. Call `sh_c_initialize_databases` function with `block_offset = 8`.
3. Call `sh_c_make_dseg` function with `path = “/”` and `node = 1`.
4. Create 9,000 byte buffer in memory and fill with 9,000 *a* characters; assign pointer `buf_ptr1` to buffer.
5. Call `sh_c_write_dseg` function with `handle = 0`, `buffer = buf_ptr1`, `byte_count = 9000`, and `offset = 0`.
6. Call `sh_c_close_dseg` function with `handle = 0`.

LPSK test program (argument value: `TEST_I3`):

7. Call `sh_i_initialize_databases` function with `block_offset = 8` and `num_dsegs = 1`.
8. Call `sh_i_get_handle` function with `path = “/”` and `node = 1`.
9. Create 9,000 byte buffer in memory; assign pointer `buf_ptr1` to buffer.
10. Call `sh_i_read_in` function with `handle = 0` and `buffer = buf_ptr1`.
11. Confirm no error returned; confirm contents of buffer matches data written to the data segment created in step 4 (test I3A).
12. Call `sh_i_read_in` function with `handle = 1` and `buffer = buf_ptr1.`
13. Confirm error `SH_C_HANDLE_INVALID` returned (test I3B).

End

`sh_i_get_dseg_size` interface test procedure: These tests correspond to Test Group 4 as described in Table 30.

Begin

Configuration test program (argument value: `TEST_I4`):

1. Call format function.
2. Call `sh_c_initialize_databases` function with `block_offset = 8`.
3. Call `sh_c_make_dseg` function with `path = "/"` and `node = 1`.

LPSK test program (argument value: `TEST_I4`):

4. Call `sh_i_initialize_databases` function with `block_offset = 8` and `num_dsegs = 1`.
5. Call `sh_i_get_handle` function with `path = "/"` and `node = 1`.
6. Call `sh_i_get_dseg_size` function with `handle = 0`.
7. Confirm no error returned; confirm 9,000 size output returned (test I4A).
8. Call `sh_i_get_dseg_size` function with `handle = 1`.
9. Confirm error `SH_C_HANDLE_INVALID` returned (test I4B).

End

`sh_i_get_dseg_hash` interface test procedure: These tests correspond to Test Group 5 as described in Table 31.

Begin

Configuration test program (argument value: `TEST_I5`):

1. Call `sh_c_format` function.
2. Call `sh_c_initialize_databases` function with `block_offset = 8`.
3. Call `sh_c_make_dseg` function with `path = "/"` and `node = 1`.
4. Call `sh_c_open_dseg` function with `path = "/"` and `node = 1`; handle zero returned.
5. Create 9,000 byte buffer in memory and fill with 9,000 *a* character; assign pointer `buf_ptr1` to buffer.
6. Generate and store hash of buffer.
7. Call `sh_c_write_dseg` function with `handle = 0`, `buffer = buf_ptr1`, `byte_count = 9000`, and `offset = 0`.
8. Call `sh_c_close_dseg` function with `handle = 0`.

LPSK test program (argument value: `TEST_I5`):

9. Call `sh_i_initialize_databases` function with `block_offset = 8` and `num_dsegs = 1`.
10. Call `sh_i_get_handle` function with `path = "/"` and `node = 1`.
11. Call `sh_i_get_dseg_hash` function with `handle = 0`.
12. Confirm no error returned; confirm returned hash matches previously generated hash (test I5A).
13. Call `sh_i_get_dseg_hash` function with `handle = 1`.
14. Confirm error `SH_C_HANDLE_INVALID` returned (test I5B).

End

3. Runtime Interfaces

Runtime interface test procedures require use of both configuration and LPSK test tools described in Chapter IV. Test procedures indicate which test program they use to perform which steps of the procedure Some tests procedures are performed immediately after previous test procedures with no break in program execution. Instructions to continue to the next test procedure are appended to these test procedures.

`sh_r_read_in` interface test procedure: These tests correspond to Test Group 1 as described in Table 32.

Begin

Configuration test program (argument value: `TEST_R1`):

1. Call `sh_c_format` function.
2. Call `sh_c_initialize_databases` function with `block_offset = 8`.
3. Call `sh_c_make_dseg` function with `path = "/"`, `node = 1`, and `size = 9,000`.
4. Create 9,000 byte buffer in memory and fill with 9,000 *a* characters; assign pointer `buf_ptr1` to buffer.
5. Call `sh_c_write_dseg` function with `handle = 0`, `buffer = buf_ptr1`, `byte_count = 9000`, and `offset = 0`.
6. Call `sh_c_close_dseg` function with `handle = 0`.

LPSK test program (argument value: `TEST_R1`):

7. Call `sh_i_initialize_databases` function with `block_offset = 8` and `num_dsegs = 1`.
8. Call `sh_i_get_handle` function with `path = "/"` and `node = 1`.
9. Create 9,000 byte buffer in memory; assign pointer `buf_ptr1` to buffer.
10. Call `sh_r_read_in` function with `handle = 0` and `buffer = buf_ptr1`.
11. Confirm no error returned; confirm contents of buffer matches data written to the data segment created in step 4 (test R1A).
12. Call `sh_r_read_in` function with `handle = 1` and `buffer = buf_ptr1.`
13. Confirm error `SH_C_HANDLE_INVALID` returned (test R1B).

Continue to next test procedure

`sh_r_write_out` interface test procedure: These tests correspond to Test Group 2 as described in Table 33.

Begin

LPSK test program (argument value: `TEST_R2`):

1. Generate and store the hash of buffer created and written to in steps 9 and 10 of the `sh_r_read_in` test procedure.
2. Overwrite the contents of the buffer with new data.
3. Generate and store the hash of the buffer.
4. Call write out function with `handle = 0` and `buffer = buf_ptr1`.
5. Confirm no error return returned (test R2A).
6. Call write out function with `handle = 0` and `buffer = buf_ptr1`.
7. Confirm error `SH_C_HANDLE_INVALID` returned (test R2B).

Continue to next test procedure

`sh_r_get_dseg_hash` interface test procedure: These tests correspond to Test Group 3 as described in Table 34.

Begin

LPSK test program (argument value: `TEST_R3`):

1. Call `sh_r_get_dseg_hash` function with `handle = 0`.

2. Confirm no error returned; compare hash value compared to hash generated in step 3 of the `sh_r_write_out` test procedure (test R3A).
3. Call `sh_r_get_dseg_hash` function with `handle = 1`.
4. Confirm error `SH_C_HANDLE_INVALID` returned (test R3B).

End

`sh_r_get_dseg_size` interface test procedure: These tests correspond to Test Group 4 as described in Table 35.

Begin

Configuration test program (argument value: `TEST_R4`):

1. Call `sh_c_format` function.
2. Call `sh_c_initialize_databases` function with `block_offset = 8`.
3. Call `sh_c_make_dseg` function with `path = “/”`, `node = 1`, and `size = 9,000`.

LPSK test program (argument value: `TEST_R4`):

4. Call `sh_i_initialize_databases` function with `block_offset = 8` and `num_dsegs = 1`.
5. Call `sh_i_get_handle` function with `path = “/”` and `node = 1`.
6. Call `sh_r_get_dseg_size` function with `handle = 0`.
7. Confirm no error returned; 9,000 size output returned (test R4A).
8. Call `sh_r_get_dseg_size` function with `handle = 1`.
9. Confirm error SH_C_HANDLE_INVALID returned (test R4A).

End

`sh_r_check_hash` interface test procedure: These tests correspond to Test Group 5 as described in Table 36.

Begin

Configuration test program (argument value: `TEST_R5`):

1. Call `sh_c_format` function.
2. Call `sh_c_initialize_databases` function with `block_offset = 8`.
3. Call `sh_c_make_dseg` function with `path = “/”`, `node = 1`, and `size = 9,000`.

4. Call `sh_c_open_dseg` function with `path = “/”` and `node = 1`; handle zero returned.
5. Create 9,000 byte buffer in memory and fill with 9,000 *a* characters; assign pointer `buf_ptr1` to buffer.
6. Generate and store hash of buffer.
7. Call `sh_c_write_dseg` function with `handle = 0`, `buffer = buf_ptr1`, `byte_count = 9000`, and `offset = 0`.
8. Call `sh_c_close_dseg` function with `handle = 0`.

LPSK test program (argument value: `TEST_R6`):

10. Call `sh_i_initialize_databases` function with `block_offset = 8` and `num_dsegs = 1`.
11. Call `sh_i_get_handle` function with `path = “/”` and `node = 1`.
12. Call `sh_r_check_hash` function with `handle = 0`.
13. Confirm no error returned; confirm true value returned (test R5A).
14. Call sh_r_check_hash_function with `handle = 1`.
15. Confirm error `SH_C_NODE_NOT_DSEG` returned (test R5B).

<u>**End**</u>

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] C. E. Irvine, T. E. Levin, T. D. Nguyen, and G. W. Dinolt, “The Trusted Computing Exemplar Project,” in *Proceedings of the 5th IEEE Systems*, (Military Academy, West Point, NY), pp. 109–115, IEEE Computer Society Press, June 2004.

[2] P. Myers, *Subversion: the Neglected Aspect of Computer Security*, M.S. thesis, Naval Postgraduate School, Monterey, CA 1980.

[3] E. A. Anderson, C. E. Irvine, and R. R. Schell, “Subversion as a Threat in Information Warfare,” 2004.

[4] C. E. Irvine, “Security: Where testing fails,” in *International Test and Evaluation Association Journal*, 21(2), pp. 53–57, 2000.

[5] Common Criteria for Information Technology Security Evaluation, Part 3: Security assurance requirements, Version 2.1, August 1999.

[6] M. D. Schroeder, “Engineering a security kernel for MULTICS,” in *Fifth Symposium on Operating Systems Principles*, pp. 125–132, 1975.

[7] S. R. Ames, M. Gasser, R. R. Schell, “Security Kernel Design and Implementation: An Introduction,” in *Computer 16*, 7, pp. 14–22, 1983.

[8] J. Rushby, “The design and verification of secure systems,” in *8th ACM Symposium on Operating System Principles 15,* 5, pp. 12–21, 1981.

[9] U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, Version 1.03, Information Assurance Directorate, 2007.

[10] C. E. Irvine, T. E. Levin, and T. D. Nguyen, “Least Privilege in Separation Kernels,” in *Proceedings 12th European Symposium on Programming, (ESOP),* pp. 159–173, 2006.

[11] M. Ward, “Hard drive evolution could hit Microsoft XP users,” *BBC News,* April 9, 2010. [Online]. Available: http://news.bbc.co.uk/2/hi/8557144.stm. [Accessed March 15, 2010].

[12] C. Ruemmler and J. Wilkes, “An Introduction to Disk Drive Modeling,” in *IEEE Computer*, 27(3), pp. 17–29, March 1994.

[13] D. Golden and M. Pechura, “The structure of microcomputer file systems,” in *Communications of the ACM,* 29(3), pp. 222–230, 1986.

[14] C. E. Irvine, T. E. Levin, T. D. Nguyen, P. C. Clark, and D. J. Shifflett, "Trusted Computing Exemplar Least Privilege Separation Kernel Product Functional Specification," Naval Postgraduate School Center for Information Systems Security Studies and Research, 2009.

[15] Software Development Standards Citation.

[16] Open Watcom Main Page [Online]. Available: http://openwatcom.org. [Accessed June 6, 2010].

[17] GCC, the GNU Compiler Collection. [Online]. Available: http://gcc.gnu.org. [Accessed June 6, 2010].

[18] Open Watcom C/C++ User's Guide. [Online]. Available: http://openwatcom.org/ftp/manuals/current/cguide.pdf. [Accessed May 28, 2010].

[19] Security-Enhanced Linux Research Site. [Online]. Available: http://nsa.gov/research/selinux/index.shtml. [Accessed June 8, 2010]. National Security Agency.

[20] XTS-400 Trusted Computer System. [Online]. Available: http://baesystems.com/ProductsServices/bae_prod_csit_xts400.html. [Accessed June 8, 2010].

[21] XTS-400/STOP 6.4 U4 Validation Results. [Online]. Available: http://www.niap-ccevs.org/cc-scheme/st/vid10293. [Accessed June 8, 2010]. National Information Assurance Partnership.

[22] Final Evaluation Report, Gemini Computers, Incorporated, Gemini Trusted Network Processor Version 1.01, National Security Agency, June 1995.

[23] P. J. Denning, S.C. Schwartz, "Properties of the working-set-model." In *Communications of the ACM* , 14(3), pp. 191–198, 1972.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Ernie Brickell
   Intel
   Hillsboro, OR

4. Kris Britton
   National Security Agency
   Fort Meade, MD

1. John Campbell
   National Security Agency
   Fort Meade, MD

2. Deborah Cooper
   DC Associates, LLC
   Roslyn, VA

3. George Cox
   Intel
   Hillsboro, OR

4. Grace Crowder
   NSA
   Fort Meade, MD

5. Louise Davidson
   National Geospatial Agency
   Bethesda, MD

6. Vincent J. DiMaria
   National Security Agency
   Fort Meade, MD

7. Rob Dobry
   NSA
   Fort Meade, MD

8. Jennifer Guild
SPAWAR
Charleston, SC

9. CDR Scott Heller
SPAWAR
Charleston, SC

10. Dr. Steven King
ODUSD
Washington, DC

11. Steve LaFountain
NSA
Fort Meade, MD

12. Dr. Greg Larson
IDA
Alexandria, VA

13. Dr. Carl Landwehr
National Science Foundation
Arlington, VA

14. Dr. John Monastra
Aerospace Corporation
Chantilly, VA

15. John Mildner
SPAWAR
Charleston, SC

16. Dr. Victor Piotrowski
National Science Foundation
Arlington Virginia

17. Jim Roberts
Central Intelligence Agency
Reston, VA

18. John Santos
CERDEC S&TCD Information Assurance Division
Fort Monmouth, NJ

19. Ed Schneider
IDA
Alexandria, VA

20. Mark Schneider
NSA
Fort Meade, MD

21. Keith Schwalm
Good Harbor Consulting, LLC
Washington, DC

22. Ken Shotting
NSA
Fort Meade, MD

23. Dr. Ralph Wachter
ONR
Arlington, VA

24. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, CA

25. Paul C. Clark
Naval Postgraduate School
Monterey, CA

26. Jonathan M. Guillen
Naval Postgraduate School
Monterey, CA