CSFS – A Secure File System

Ying Liu

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
At Concordia University
Montreal, Quebec, Canada

September 2003

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Ying Liu

Entitled: CSFS - A Secure File System

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

____________________________________ Chair
E. Doedel

____________________________________ Examiner
G. Grahne

____________________________________ Examiner
O. Ormandjieva

____________________________________ Supervisor
B. C. Desai

Approved by ____
Chair of Department or Graduate Program Director

_____________2003 ____
Dr. Nabil Esmail, Dean
Faculty of Engineering and Computer Sciences

ABSTRACT

CSFS - A Secure File System

Ying Liu

In the twenty-first century, people are accustomed to using computers to deal with their daily work and personal affairs. However, in an increasing number of computer applications in various fields, the security issue has become very important. By “secure”, we mean secrecy and tampering detection. Therefore, the issue of how to guaranty the security of computer applications comes up naturally. We need more secure and dependable software systems to provide stronger protection of our sensitive data. This thesis presents CINDI Secure File System (CSFS), which is based on the implementation of two secure systems. One is a simple Secure Database Management System (SGDBM), which deal with data and metadata uniformly. The Secure DBMS is based on the GNU Database Management System (GDBM) in Linux. The GDBM is transformed to be a secure database system by embedding encryption and hashing. The other is a Secure File System that is based on SGDBM previously mentioned. CSFS uses the SGDBM to store file passwords for the encrypted files. It provides a number of commands, which can be used in the same way as other basic commands in Linux, and a simple GUI application for the secure file operations.

ACKNOWLEDGMENTS

I’d like to first acknowledge my appreciation to my thesis supervisor Bipin C. Desai who provided me with many useful directions and suggestions so that I can accomplish my thesis smoothly. In particular, my thesis topic suggested by my supervisor gave me a good opportunity to combine knowledge from various computer fields and provided a reason for careful and in-depth study. I learned so much during the process of completing my thesis and feel more confident of my future. Secondly, I am grateful to my parents and twin sister who have given me consistent love and support especially when I encountered difficulties. In addition, I would like to thank all the professors who are in charge of examining my thesis. Their guidance and help has improved my thesis.

TABLE OF CONTENTS

TABLE OF FIGURES

1. Introduction

1.1 Introduction

In the twenty-first century, people all over the world are getting used to using computers and computer networks to deal with their daily work and personal affairs. However, with an increasing number of computer applications in various fields, the security issue has become very important. Secure here means both secrecy (protection against reading from un-trusted programs) and tampering detection (protection against writing from un-trusted programs). Nobody would like his or her personal and confidential information to be exposed to other people. For governments and enterprises, there is also much data that has to be kept secure and cannot be accessed by any unauthorized person. Therefore, the issue of how to guarantee the security of computer applications came up naturally.

A large number of computer applications involve database operations. The security issue of databases has attracted wider attention. Most of the enterprises, involved in the development of database management system, have endeavored to make their database systems more secure and more reliable. Usually, most of them choose to add a specific layer on the top of their original database management system to provide some encryption and validity checks.

Some emerging computer applications require programs to maintain sensitive state on un-trusted computers. Therefore, more secure and dependable software systems are expected to provide stronger protection of sensitive data. Some new technologies were developed accordingly. One of them is the Secure Database Management System. It integrates encryption and hashing with a low-level data model, which can protect data and metadata uniformly, unlike other systems built on top of conventional database systems.

People might consider that building a Secure Database Management System by layering cryptography on top of a conventional database system is enough [Mich00]. This layer could encrypt objects before storing them in the database and maintain a tree of hash values over them. This architecture is attractive because it can be easily implemented and does not require building a new database management system. Unfortunately, this layer is not able to protect the metadata inside the database system. For example, an attack is possible to effectively delete an object from the database by modifying the indexes. There could be some performance problems as well. For example, the database system could not maintain ordered indexes for range queries on encrypted data. For these reasons, Secure Database Management System applies encryption and hashing to a low-level data model, which protects both data and metadata in the same way. In addition, in secure DBMS, the database overhead is mostly dominated by disk I/O. Encryption and hashing operations represent only a little of the total overhead.

Another technology developed to keep data secure is the Secure File System. SFS provides transparent access to encrypted files and protects the sensitive data even when the media is stolen. It usually works on top of a conventional file system.

In this project, two secure systems have been implemented. One is an elementary Secure Database Management System: SGDBM. It transformed the existing GDBM system in

Linux to be a Secure DBMS by embedding encryption and hashing algorithm. It accomplished the object to make both data and metadata secure uniformly.

The other secure system is a completely new layer of security to provide a Secure File System (CSFS). CSFS provides necessary secure file operations. CSFS is implemented by adding a new layer on the conventional file system of Linux. It also used cryptography and hashing algorithm to accomplish the security objective.

CSFS depends on SGDBM because it uses SGDBM to do all necessary database operations. In CSFS, the key used to encrypt a file is stored into the SGDBM database. Even though a hacker obtains the database file, he cannot find out the key for the encrypted file because the data in the SGDBM database is also encrypted. In other words, SGDBM provides reading protection. To decrypt a file, it fetches the key from the database and checks if the key has been tampered with before decrypting the file. SGDBM can provide tampering detection because it is a secure database system. If CSFS uses a conventional database management system, such as GDBM, it cannot tell if the key has been tampered because the conventional DBMS does not provide tampering detection.

1.2 Report Outline

The rest of this report is organized as follows. Chapter two discusses some existing secure systems. Chapter three presents the basic technical background details on which SGDBM and CSFS are based. Chapter four describes the overall architecture and implementation of both of SGDBM and CSFS. In chapter five, a manual gives the method to install and use the secure system. Chapter six gives the contribution and conclusion of this thesis, presents some existing problems and future work.

2. Related Technology Introduction

In this chapter, some existing Secure DBMS and Secure File Systems will be simply presented and the difference between their implementation and SGDBM and CSFS will be analyzed.

2.1 Other Secure DBMS

There are not many implementations of Secure Database Management System. A couple of them will be simply described below.

In the technical report: “How to Build a Trusted Database System on Untrusted Storage” written by Umesh Maheshwari, Radek Vingralek and William Shapiro; they came up with an architecture and implementation of a Secure Database Management System [UMES02]. They called it a trusted database system, TDB. This system leverages a trusted processing environment and a small amount of trusted storage available on the platform and provides security by encrypting data with a key hidden in this storage. TDB stores a collision-resistant hash of the database in tamper-resistant storage.

TDB applies hashing and encryption to a low-level data model, which protects data and metadata uniformly. It also enables TDB to maintain ordered indexes on data. To protect the sensitive state from media failures such as disk crashes, TDB provides the ability to create backups and to restore valid backups. TDB uses log-structured storage systems and exploits the synergy between the functions for backups and log-structured systems. The implementation of this Secure DBMS requires some hardware support. Compared with

TDB, SGDBM implemented in this project is a complete software solution without the necessity of any extra hardware.

Conventional database management systems (DBMS) do not recognize different security levels of data they store and retrieve because they treat all data at the same security level. Multi-level secure (MLS) DBMS schemes put forward by the Carnegie Mellon Software Engineering Institute provide a means of maintaining a collection of data with mixed security levels [MLS00]. The access mechanisms allow users or programs with different levels of security clearance to store and obtain only the data appropriate to their level. Multi-level secure DBMS architecture schemes are categorized into two general types: the Trusted Subject architecture and the Woods Hole architectures.

The Trusted Subject architecture is a scheme that contains a trusted DBMS and an operating system [MSL00]. The DBMS is custom-developed with all the required security policy (the security rules that must be enforced) developed in the DBMS itself. The DBMS uses the associated trusted operating system to make actual disk data accesses. This scheme results in a special purpose DBMS and operating system that requires a large amount of trusted code to be developed and verified along with the normal DBMS features. A benefit of the Trusted Subject architecture is that the DBMS has access to all levels of data at the same time, which minimizes the retrieval and update process. This scheme also can handle a wide range of sensitivity labels and supports complex access control.

The Woods Hole architectures assume that an un-trusted (usually commercial-off-the-shelf (COTS)) DBMS is used to access data and that trusted code is developed around that DBMS to provide an overall secure DBMS system. The three different Woods Hole architectures address three different ways to wrap code around the un-trusted DBMS.

The Distributed architecture scheme uses multiple copies of the trusted front end and DBMS, each associated with its own database storage. In this architecture scheme, data is replicated in the common database. When data is retrieved, the DBMS retrieves it only from its own database. A benefit of this architecture is that data is physically separated into separate hardware databases.

Another Secure Database Management System was developed by several Ph.D students of National University of Singapore [HWEE03]. They developed three strategies to create a Secure DBMS. The first is Secret-Sharing DBMS. One way to strengthen a database is to employ a secret sharing scheme or an information dispersal scheme, where the database is split into multiple parts under different administrators' control so that a certain quorum is required to reconstruct the original content. The quorum can be set to achieve an optimal balance – a higher quorum reduces the likelihood of data leakage, whereas a lower quorum provides more redundancy for countering denial of service attacks.

While the secret sharing and information dispersal schemes enhance the security of databases, this comes at the expense of additional storage space and runtime overhead in reconstructing the original data. At present hardware prices, storage overhead is not likely to be deterrence. However, runtime overhead must be managed carefully, especially when each data operation accesses multiple shares over a wide-area network such as the Internet [HWEE03].

The second approach used is Steganographic DBMS. In order to resist any compulsion to divulge valuable data, it is necessary to make sure that the attackers are not able to establish such in the first place. Steganography, the art of hiding information in ways that prevent its detection, offers plausible deniability to the user or administrator. It is a better defense against compulsion than cryptography – While cryptography scrambles a message so it cannot be understood, the cipher text itself is likely to alert the attacker to the existence of protected data [HWEE03].

This strategy aims to develop a Steganographic DBMS that allows a user to associate a password with each data set, so that subsequent transactions on any data succeed only when the correct password is supplied. An attacker who does not have the password can get no information about whether the data set even exists in the DBMS.

The third strategy they proposed is Anonymous DBMS. When a DBMS resides on publicly accessible servers, e.g. in a peer-to-peer network or with a third-party service provider, to support remote users and applications, securing the DBMS alone is inadequate. To ensure high availability in the face of such potential threats, a remote

DBMS needs to be augmented with extra security mechanisms beyond conventional data replication and recovery [HWEE03].

The objective is to design such an anonymous DBMS. It will allow a sequence of authorization codes to be associated with each (group of) data items, and the DBMS will honor only requests that are accompanied by the expected authorization code within the sequence. Since the users and applications do not know where the DBMS is, the requests either have to be forwarded by a trusted proxy, or the DBMS has to pick them up at a pre-arranged location. Furthermore, the database will be physically split across several DBMS sites, for example within a peer-to-peer platform on the Internet, in such a way that the copy on any given site reveals nothing about the original data. This safeguards the operator of the DBMS sites from being incriminated for storing any controversial content [HWEE03].

2.2 Other Secure File System

Secure File Systems [SFS00] provide encryption and decryption of files to make files secure. Generally, there are two approaches to implement a secure file system.

The first one is to modify the kernel file system implementation. Its advantages include the speed of implementation, usage of internal kernel structures and simpler implementation of some operations. But it has also some disadvantages. First of all, it requires modifications of the kernel of the operating system; this may cause instability that is very hard to debug.

The second approach is to implement the Secure File System in the user space. This way, it is not necessary to touch the original operating system and its file system. Just a layer will be added on the top of the conventional file system. The disadvantage is that users have to run some special programs to use the Secure File System.

There is a GNU secure file system distributed under GNU GPL [GPL00] and can be downloaded as a RPM package, SPRM package or tgz package [MICH00]. It uses the second approach. It works on top of a normal file system and using its own library redirects all file operation requests. It is implemented by providing a daemon (*sfsd*), which does most of the work for SFS, and the shared library *libsfs*, that is, in fact an envelope for the standard file operations such as open, read or write that provides additional functionality – encryption. Even though implemented in the user space, this implementation of secure file system remains completely transparent to the users as long as the shared library is available and the daemon is run on the computers. Using this approach, the *libsfs* library just takes care of forwarding requests to the daemon *sfsd* that does the real work of encryption and similar tasks. The communication between *sfsd* and library functions is done through message passing. So *sfsd* listens on one permanent message queue for incoming requests.

In the GNU SFS system, each file is encrypted by a file key generated randomly and then the file key is encrypted by a public key and then saved into some relative files with prefix '.'. When the user needs to decrypt a file, the user's login password is used as private key to decrypt the corresponding file key and then decrypt the file using decrypted file key. The major advantage of this approach is that it is completely transparent to users. The GNU secure file system does not use database and has some limitations. The major disadvantage is that the files used to save file keys cannot be protected very well. In addition, it is not able to deal with any tampering of the file content because this system does not provide any method to check if the decrypted file content is the same as its plain file content.

Compared with the above approach, the CSFS proposed herein has overcome the major disadvantages of the existing secure file systems. However, the proposed system is not completely transparent to users. Users have to obviously come up with requirements by executing appropriate commands or a simple GUI application provided.

In addition to the secure file system presented above, there are still other implementations of secure file system. One example is the Satan file system developed at Carnegie-Mellon University. The implementation employs C library modifications that read a file into main memory, decrypt the data and then deliver it to the application [GART97]. The basic idea is to link the applications against a set of libraries that provide encrypted versions of standard library calls. This solves the problem of rewriting all the applications, but the applications still need to be recompiled or at least re-linked. Also every application program will have to have an unencrypted and encrypted version for working with encrypted files or unencrypted files.

Microsoft’s Encrypted File System (EFS) is included with the Microsoft Windows 2000 operating system. It is based on public-key encryption and takes advantage of the CryptoAPI architecture in Windows 2000. Each file is encrypted using a randomly generated file encryption key. File encryption and decryption is supported per file or for an entire folder. Folder encryption is transparently enforced. All files (and folders) created in a folder marked for encryption are automatically encrypted. Each file has a unique file encryption key. Users do not have to decrypt a file to open it and use it. The EFS automatically detects an encrypted file and locates a user’s file encryption key from the system’s key store. Command-line tools and administrative interfaces are provided for users. This is not a quiet secure system because it cannot prevent hackers’ access to files once they have access to the computers.

3. State of Arts Survey

To accomplish CSFS and SGDBM, some existing algorithms and technologies, including CPUID, Blowfish, SHA-1 and GDBM in Linux, were studied and used. Some survey and basic concepts of these technologies will be given in the following sections in order for the readers to understand the implementation of both secure systems more easily.

3.1 Blowfish

In this project, Blowfish was chosen to for encryption to provide reading protection for both SGDBM and CSFS. There are many different encryption algorithms available currently. However, most of these algorithms are unavailable to the public. Blowfish was released to the public domain as a free alternative to existing encryption algorithms. Blowfish is not patented and license-free, and is available free for all uses. This is the main reason to choose Blowfish in this project. The comparisons between Blowfish and other cryptographies can be found in the referenced paper [EAC01] and [EAC02].

The Blowfish algorithm was designed by Bruce Schneier in 1993 [SCHN93]. It is slowly gaining acceptance as a good encryption algorithm. Blowfish is a symmetric 64-bit block cipher. It takes a variable-length key, from 32 bits to 448 bits (56 bytes), making it ideal for both domestic and exportable use. The algorithm consists of two parts: a key-expansion part and a data-encryption part. Key expansion converts a key of at most 448 bits into several subkey arrays totaling 4168 bytes in fixed length. These sub-keys must be pre-computed and generated before any data encryption or decryption.

Data encryption occurs via a 16-round Feistel network. Each round consists of a key-dependent permutation, and a key- and data-dependent substitution. All operations are XORs and additions on 32-bit words. The only additional operations are four indexed array data lookups per round.

Because the Blowfish process 64-bit blocks, the data must be a multiple of 64 bits (8 bytes) in length. If not, some padding must be done. The encrypted data always has the same length as the original plain data.

3.2 Writing BIOS

As mentioned in the above section, Blowfish takes a variable-length key, from 32 bits to 448 bits (56 bytes). How to generate the key used by Blowfish has been considered carefully. At the beginning of the period of the system design, it was suggested that we tried to store one part of the key in the unused space in a BIOS chip. We tried to implement this work but the attempt was unsuccessful. One of the reasons for this was the lack of the documentation in BIOS and a wide variety of these for each generation of PC.

An intensive search for the most up-to-date information was made on the Internet and some attempts were made to implement using BIOS but without success. What was tried will be presented below in order that this problem can be discussed in the future work. In theory, the data in a BIOS chip is not allowed to change randomly by users. In the past, the BIOS reside in either EPROM or EEPROM chip that can be rewritten only by using a specific hardware [BIOS00]. Recently, with the speedy progress of technology, flash memory enables users to rewrite the BIOS chip using a software solution. All the manufactures of BIOS chips provide their special software utilities to enable users to update their BIOS when necessary. However, these special programs belong to the manufactures that developed them, so they are not free and open source. Thus, an approach is needed to write a few data in the unused space in a BIOS chip.

We studied the PC memory allocation rules and noted that the BIOS data always occupied the address F0000 to FFFFF of the system memory space [BIOS2]. Hence the data to be used in the lees should be stored into the BIOS in free space within this memory address space. However, when we read the BIOS data and saved it as a binary file using the special software utility provided by the manufacture, we found that the size of this binary file is far more than the 64K bytes it occupies in the system memory.

We debugged this binary file and tried to find data matches between the BIOS memory and its copy as stored as a binary file, but we only found some matching readable text such as the manufacture and year of the BIOS. There was no technical literature available to do this job and all attempts were made by guesswork. We found some seemingly unused space in the binary file and wrote some test data into that space. Unfortunately, when we wrote the modified BIOS binary file back to replace the original BIOS, it failed and the computer would no longer boot. We could not even restore the BIOS to the original state so we had to terminate attempts. However, we believe that there should be a way to resolve the problem, if the technical details of the various BIOS are available.

3.3 CPUID and Processor Serial Number (PSN)

Processor Serial Number was chosen to be a part of the key used by Blowfish in both SGDBM and CSFS system. It can uniquely identify CPUs so that both secure systems can make sure that each computer will have a unique key for encryption. The PSN is returned by CPUID instruction. CPUID and PSN will be presented below.

In the earlier days of x86 computing, there was a large amount of different hardware inside PCs, much as there is today, but back in those days "compatibility" was an unknown concept between manufacturers - all their hardware behaved differently, and the poor programmer had to write mountains of code merely to identify the hardware the program was running on, yet without taking into account the work involved to take full advantage of it's unique features. The CPU was one area where, despite continual advancements by Intel, AMD and others, programs were generally not using any optimization based on the CPU, because it was difficult to know which CPU was being used [CPUID1].

When Intel released MMX, however, it decided that it needed to make it easy for programs to recognize it's new CPUs, and utilize the instructions provided to increase the performance of the programs when running under this CPU. So, they developed the CPUID instruction. This simple assembler instruction gave the programmers instant access to a lot of information regarding the CPUs: the manufacture of the CPU, what "extra features" it supports, and other useful information.

Generally speaking, virtually all Pentium CPUs support the CPUID instruction. Also, genuine Intel 486-based CPUs and many 486 clones support it. The Pentium Pro, Pentium II and Pentium III all support it. All AMD**'s** CPUs support it, and the Cyrix MediaGX, 6x86, 6x86MX and mII. Almost all CPUs available today support CPUID. Unfortunately, there is a small problem with just calling CPUID to find the CPU information. If the CPU you are running on does not support CPUID, it may crash. Before trying to rely upon CPUID, a program should properly detect and sometimes enable the instruction. In particular, the program must detect the presence of a 32-bit IA-32 processor, which supports the EFLAGS register. Next, if it is a Cyrix or a NexGen processor, the CPUID instruction may have to be enabled. Then the program must try to toggle the ID bit (bit 21) in the EFLAGS register, to determine whether the instruction is supported or not [CPUID2].

The CPUID instruction is very easy to call from any language that supports in-line assembler. The CPUID instruction does not have any parameter; instead the value of the internal register *eax* is set before calling it. The value set in *eax* determines which kind of information you need and which CPUID function you want to call. The result information related to different function is returned in other registers such as *ebx*, *ecx* and *edx*. Note that all processors handle *eax* equaling to 0 the same way, but after that different manufacturers return different information [CPUID1].

With the Pentium III, Intel introduced a new feature called the processor serial number (PSN) [PSN01]. This is a feature that, according to Intel, would usher in the next generation of software applications for the world of networked computing. The processor serial number (PSN) is a unique identifier for an individual microprocessor that cannot be modified, but can be read by software to provide identification of a processor.

The PSN is a 96-bit number programmed into the processor core at manufacturing time. The number is programmed into the silicon of the processor and cannot be modified. The upper 32 bits of the PSN provide coded information of the processor family type. This is currently read by the CPUID instruction on all Intel and Intel-compatible processors, regardless of whether the processor contains the PSN feature or not. The lower 64 bits, however, are different for all PSN-capable processors, providing a unique identifier with no independent meaning.

Two new instructions are associated with the PSN: a read instruction and a disable instruction [PSN01]. The read instruction returns the full 96-bit PSN when allowed. A Model Specific Register (MSR) bit, controlled by the disable instruction, determines whether the PSN can be read or not. If the bit is set to 0, the full PSN can be read by software. If it is set to 1, the ability to read the PSN is disabled, and only the non-unique 32-bit CPUID is readable; the bottom 64 bits of the PSN remain undefined during the read.

Once the PSN is disabled using the disable instruction, a hardware reset of the processor (i.e. a machine boot up, or in some systems a resume from deep sleep) is necessary to allow reading of the PSN. This design was intended to protect the privacy of the user so that he or she would be aware if the PSN was being made readable again. The user would knowingly have to reset the machine to turn on the PSN. An outsider theoretically should not be able to enable reading of the PSN without the user knowing. By default, Linux disable the PSN when booting the kernel.

3.4 SHA-1

In both SGDBM and CSFS, to provide tampering detection, a type of hashing algorithm is required. The Secure Hash Algorithm-1 (SHA-1) was chosen in this project because it is generally accepted and available free without licensing.

SHA-1 is a United States Department of Commerce National Institute of Standards and Technology approved standard (FIPS Pub 180-1) for secure hashing [SHA195]. The SHA-1 is designed to work with the Digital Signature Standard (DSS). The Secure Hash Algorithm (SHA-1) is necessary to ensure the security of the Digital Signature Standard. When a message of length less than 2^64 bits is inputted, the SHA-1 produces a 160-bit representation of the message called the message digest. The message digest is used during generation of a signature for the message.

The SHA-1 is designed to have the following properties: it is computationally infeasible to recover a message corresponding to a given message digest, or to find two different messages which produce the same message digest. The message or data file should be considered to be a bit string. The length of the message is the number of the bits in the message. If the length of the message is not a multiple of 512 bits (64 bytes), some padding must be done because SHA-1 sequentially processes blocks of 512 bits when computing the message digest.

3.5 GDBM in Linux

SGDBM is derived from GNU database management system (GDBM) in Linux. SGDBM transformed GDBM by providing reading protection and tampering detection. In this project, GDBM was chosen to be a basis of implementing SGDBM because it is open source and suitable for experimentation.

GDBM is a library of database functions that use extendible hashing and works similar to the standard UNIX dbm functions [GDBM03]. These routines are provided to a programmer needing to create and manipulate a hashed database. (The GDBM is not a complete database package for an end user.) The basic use of GDBM is to store key / data pairs in a data file. Each key must be unique and each key is paired with only one data item. The keys cannot be directly accessed in sorted order. The basic unit of data in GDBM is the structure:

```
typedef struct{
    char * dptr;
    int dsize;
}datum;
```

This structure allows for arbitrary sized keys and data items.

Key / data pairs are stored in a GDBM data file, called a GDBM database. An application must open a GDBM database to be able to manipulate the keys and data contained in the database. The GDBM allows an application to have multiple databases opened at the same time. When an application opens a GDBM database, it is designated as a reader or a writer. A GDBM database can be opened by at most one writer at a time. However, many readers may open the database simultaneously. Readers and writers cannot open the GDBM database at the same time. GDBM is implemented as a library and provides some functions for programmers to call. The implementation of GDBM will be presented in the next chapter.

4. CSFS Implementation

The CSFS implementation consists of a number of parts. Two major parts are the implementation of the secure database system, SGDBM, and the implementation of a secure file system. Other parts include the implementation of code to generate the key for encryption and decryption and a simple GUI application. The detailed description of implementation for each part follows.

4.1 Key Generation Implementation

To make data secure, some form of cryptography is necessary. Blowfish is used, in both SGDBM and CSFS systems, as the encryption scheme. Blowfish is a type of cipher using the same key for both encryption and decryption. The length of the key used by Blowfish is variable and must be less then 448 bits (56 bytes). The key generations for SGDBM and CSFS are different.

4.1.1 Key Generation for SGDBM

In SGDBM, all data including both metadata and data is encrypted using Blowfish. Each user has his own SGDBM database with his own unique key for Blowfish. The key used by Blowfish in SGDBM is fixed for each unique user and generated automatically by system whenever the user’s database is accessed. The key consists of three parts: the encrypted user’s login password, the serial number of the first IDE hard disk and the processor serial number (PSN) of the CPU currently used. The combination of these three parts can uniquely identify a key for each individual user. The two parts relevant to hardware can be used to identify a unique computer. The encrypted user's login password can be used to identify each individual user logging in the computer so that we can make sure that each individual user has his unique key for Blowfish used in SGDBM.

Before introducing these three parts of a key in detail in the following sections, an important thing has to be mentioned. In this system, the key for Blowfish used in a SGDBM is fixed, that is to say, cannot be changed by any user directly. The key is generated automatically whenever it is required. The key is designed to be invariable in a SGDBM because that the SGDBM is used only by CSFS but not the users. If need be, the components of a key for Blowfish can be changed accordingly.

4.1.1.1 Encrypted User's Login Password

The first part of the key for Blowfish used in a SGDBM is the encrypted user's login password. Using the user's password can make sure that each individual login user who uses a computer with other users has a personal key for his database because each user has his own SGDBM database. To obtain the encrypted user's login password, a small function was developed and it can return the encrypted user's login password as a character string. The system functions invoked to obtain the encrypted user's login password are *getuid( )* and *getpwuid( )*. These functions read the system file *passwd*, using the root's permission, to get the encrypted user's login password. For the file *passwd*, only the root has the read permission and nobody has write and execute permission. Therefore, it is safe to use the encrypted user's login password as a part of the key in SGDBM.

One thing necessary to mention here is that, in CSFS, if a user needs to change his login password; he has to open all files in CSFS first. After the password is changed, the user need to resave those files again in CSFS. The reason is that, in SGDBM, the keys used by Blowfish to encrypt and decrypt file passwords will change with the changes of user's login password.

4.1.1.2 Hard disk Serial Number

The second part of the key for Blowfish used in SGDBM is the serial number of the first IDE hard disk. The serial number is hard coded onto the hard disk and never changes. Each hard disk has a serial number, however, if two computers have just the same hard disk model and brand, the number may be the same. In SGDBM, only the serial number of the master disk on the first IDE controller (/dev/hda) is used. Other type of hard disk controllers has not been taken into account. If need be, it is easy to modify the code to extend the related function.

To obtain the hard disk serial number, a small function was developed to return this number in a character string. In Linux, there is a system function *ioctl( )* which can be used to get a structure 'hd_driveid' that contains the hard disk information. One number variable of the structure 'hd_driveid' is serial_no. It is a 20 bytes character array and holds the serial number of a hard disk. The part of the function used to get the hard disk serial number is as follows:

```
Struct hd_driveid id;
ioctl(fd, HDIO_GET_IDENTITY, &id);
memcpy(HdKey + i, &(id.serial_no[i]), 1);
```

4.1.1.3 Processor Serial Number (PSN)

The last part of the key for Blowfish used in SGDBM is the processor serial number (PSN). Each Central Processor Unit (CPU) has a unique PSN that is built into the chip. To obtain this number by programming, it is necessary to run the instruction CPUID. CPUID supports many levels to provide variable information about a processor. To get the PSN, this instruction needs to be run at level three. The PSN is usually put into two internal registers: *ecx* and *edx*. However, for some special type of CPU, it may need another internal register *ebx* to store part of the PSN. In SGDBM, just the part of the PSN stored in the *ecx* and *edx* registers is used. A small program was written in Assembly Language to obtain the PSN and store the data in *ecx* and *edx* into variables for further use.

However, the PSN is only supported and enabled when the PSN feature flag is set. By default, Linux disables the PSN when booting the kernel and it is impossible to enable the PSN by programming. To enable the PSN in Linux, the only approach is to modify the related part of the Linux kernel source code and then recompile it so that the new kernel can be bootable. The location in the kernel where the PSN is disabled by default is in the kernel source file *setup.c* in the directory */usr/src/linux/arch/i386/kernel*. The related code is:

```
static void __init squash_the_stupid_serial_number(struct cpuinfo_x86 *c)
{
    if( test_bit(X86_FEATURE_PN, &c->x86_capability) && disable_x86_serial_nr )
   {          /* Disable processor serial number */
              unsigned long lo, hi;
```

```
                rdmsr(MSR_IA32_BBL_CR_CTL,lo,hi);
                lo |= 0x200000;
                wrmsr(MSR_IA32_BBL_CR_CTL,lo,hi);
                printk(KERN_NOTICE "CPU serial number disabled.\n");
                clear_bit(X86_FEATURE_PN, &c->x86_capability);

                /* Disabling the serial number may affect the cpuid level */
                c->cpuid_level = cpuid_eax(0);
        }
    }
```

Just one line needs to be changed to make the PSN enabled at boot time. The line:

```
    lo |= 0x200000;
```

should be changed to:

```
    lo |= 0x000000;
```

After modifying the kernel, to make the new kernel take effect, the whole new kernel must be recompiled. How to recompile the new kernel will be introduced in the next chapter. After recompiling the new kernel, the related program can obtain the PSN to be the last part of the key for Blowfish in SGDBM.

4.1.2 Key Generation for CSFS

The CINDI Secure File System (CSFS) uses Blowfish for encryption and decryption. It is required to encrypt not only all the files in CSFS but also all the keys used to encrypt those files. These two kinds of keys are generated in different ways and are going to be presented in the following sections.

4.1.2.1 Keys Used to Encrypt Files

In CSFS, each file needs to be encrypted by Blowfish. Keys used to encrypt files are called file passwords and are directly inputted by users from the keyboard arbitrarily. The file passwords can be in any form without the necessity to keep them easy to remember, because users do not need to input them again when their files are decrypted. However, the length of each file password must be less than 448 bits to meet Blowfish's requirement. Moreover, different files could differ in their file passwords.

4.1.2.2 Keys Used to Encrypt Files Passwords

All the file passwords are encrypted by Blowfish and then stored in SGDBM databases for future use. All the keys used to encrypt file passwords are generated in the way similar to generating keys for Blowfish in SGDBM. They are made up of three parts. The first two parts are the same as those in the key generated in SGDBM. They are the secure number of the first IDE hard disk and the processor serial number (PSN) of the CPU currently used. The third part of the key is called the user password and is inputted directly by the user from the keyboard. This kind of keys, unlike the one generated in SGDBM, can be variable by giving different user passwords.

It is recommended but not required that each user use the same user password for all of his files so that it is easy to remember. The user password here should be not more than 28 bytes and can be anything easy to remember. It is recommended that users do not use the same password as their login password. If two passwords are same, once an unauthorized person finds a user's login password and login the system as this user, he is able to decrypt this user’s files using the same password. That is not safe. However, users are allowed to use different user passwords for different files as long as they are able to remember all these user passwords.

4.2 Implementation of Blowfish

Blowfish is used as the encryption scheme for both SGDBM and CSFS to encrypt and decrypt all the necessary data. One structure and five functions are developed to implement the Blowfish algorithm. They are presented below.

Structure:

```
typedef struct {
    unsigned long P[16 + 2];
    unsigned long S[4][256];
} BLOWFISH_CTX;
```

In this structure, P array consists of eighteen 32-bit sub-keys: P_1, P_2, P_3, …, P_{18}.

S is a two-dimension array consisting of four 32-bit S-Boxes and each S-Box has 256 units:

$S_{1,0}$, $S_{1,1}$, $S_{1,2}$, … $S_{1,255}$

$S_{2,0}$, $S_{2,1}$, $S_{2,2}$, … $S_{2,255}$

$S_{3,0}$, $S_{3,1}$, $S_{3,2}$, … $S_{3,255}$

$S_{4,0}$, $S_{1,1}$, $S_{1,2}$, … $S_{1,255}$

This structure is used to hold all of the sub-keys during both encrypting and decrypting by Blowfish.

Function:

```
void Blowfish_Init(BLOWFISH_CTX * ctx, char * key, int keyLen);
```

This function uses the original P-Array, S-Boxes and the key given in any length less than 448 bits to generate the 4168-byte sub-keys. The sub-keys are stored in a variable of the structure BLOWFISH_CTX.

Function:

```
void Blowfish_Encrypt(BLOWFISH_CTX * ctx, unsigned long * xl, unsigned long * xr);
void Blowfish_Decrypt(BLOWFISH_CTX * ctx, unsigned long * xl, unsigned long * xr);
```

These two functions are used to encrypt and decrypt any 64-bit block of data. Each 64-bit block is divided into two parts and is put into two 32-bit variables. The two functions use the sub-keys in the variable of the structure BLOWFISH_CTX to encrypt and decrypt the two 32-bit parts of a block and then save the changed value in the original variables.

Function:

```
void Encrypt(unsigned long * data, char * key, int blknumber);
void Decrypt(unsigned long * data, char * key, int blknumber);
```

These two functions are the interface provided to the outer systems. Users provide their keys, the data need to be encrypted or decrypted and the number of the 64-bit blocks of the data. In these two functions, the three functions mentioned right above are called to calculate the sub-keys using the given keys and encrypt or decrypt all of the 64-bit blocks of the given data.

There is a limitation in the implementation of Blowfish that the data to be encrypted or decrypted must be a multiple of 64-bit in length. Thus, the length of the data will not change after it is encrypted or decrypted. In both the SGDBM and the CSFS, all the data necessary to be encrypted or decrypted were padded to be a multiple of 64-bit in length.

4.3 Implementation of SHA-1

Like Blowfish, SHA-1 is used in both the SGDBM and the CSFS. SHA-1 is a technical revision of Secure Hash Algorithm (SHA) [SHA195]. It is used to compute a condensed representation of a message. When a message of any length less than 2^{64} bits is inputted, the SHA-1 produces a 160-bit output called a message digest. The SHA-1 is used in SGDBM to make sure it is able to detect any data in the database illegally tampered by some unauthorized person. The SHA-1 is also used in CSFS to check if either file contents or file passwords have been tampered.

The SHA-1 sequentially processes 512-bit blocks while computing the message digest. It considers the message of any length inputted by users to be a bit string. If the total number of bits in a message is not a multiple of 512, the SHA-1 uses padding to make the total length of a padded message a multiple of 512. The message digest is computed using the final padded message. To implement the SHA-1 algorithm, the following structure, macros and functions are used:

Structure:

```
#define SHA1_INPUT_BYTES   64           /* 512 bits = 64 bytes = 16 words*/
#define SHA1_INPUT_WORDS   16
#define SHA1_DIGEST_BYTES  20
#define SHA1_DIGEST_WORDS  5            /* 160 bits = 20 bytes = 5 words */

typedef struct {
    word32 H[SHA1_DIGEST_WORDS];    /* output is a 5 words (160 bits) digest */
    byte M[SHA1_INPUT_BYTES];       /* input is a 64 chars (512 bits) message */
    #if defined(word64)
       word64  bits;                    /* we want a 64 bits word used to save the size of data*/
    #else
       word32  hbits, lbits;            /* if we do not have one we simulate it */
    #endif
} SHA1_ctx;
```

In this structure, the H-Array is used to hold the 160-bit message digest. The M-Array is used to hold the 512-bit input data block of a message.

Macro:

```
#define SHA1_set_IV(ctx, IV)   memcpy((ctx)->H, IV, SHA1_DIGEST_BYTES)
………
extern word32 SHA1_IV[5];
#define SHA1_init(ctx)   SHA1_zero_bitcount(ctx); SHA1_set_IV(ctx, SHA1_IV)
#define SHA1_init(ctx)
```

These macros are used to initialize the H-Array in a variable of the structure SHA1_ctx. The H-Array keeps changing during the process of calculating the message digest.

Function:

```
void SHA1_transform(word32 H[SHA1_DIGEST_WORDS], const byte M[SHA1_INPUT_BYTES]);
```

This function uses both the old H-Array and a 512-bit data block of message as input to get a new H-Array what is going to be used for the next shift.

Function:

```
void SHA1_update(SHA1_ctx * ctx, const void * pdata, word32 data_len);
void SHA1_final(SHA1_ctx * ctx);
```

These two functions process the whole message and get the final message digest. SHA1_update function divides the character message into several 512-bit blocks and transforms them (using the previously presented function) one by one except the last block. SHA1_final function deals with the rest data of the message to be the last 512-bit block and transform it to get the final H-Array what is the message digest.

Function:

*unsigned long * sha1(char * data, int data_len);*

This function is the interface provided to the outer systems. Users provide their character string of any length as the message and the function returns a 160-bit H-Array what holds the message digest. In this function, the macros and functions mentioned right above are called to process all the 512-bit blocks within the given message.

4.4 Implementation of SGDBM

This section presents the architecture and implementation of the Secure GDBM (SGDBM). The whole database, in SGDBM, is encrypted by Blowfish and validated by SHA-1; so unauthorized users or programs cannot read the database or modify it undetectably. As mentioned previously, the implementation of SGDBM is tightly based on GDBM, an elementary database management system in Linux. Moreover, GDBM is absolutely open so that it is possible to have it modified and transformed to be a secure database management system.

All the modifications made to GDBM are twofold. One is the use of the Blowfish for encryption and decryption. The other is the use of the SHA-1 for validation. In the following sections, these modifications and the implementation of SGDBM in will be presented in detail.

4.4.1 Data Structures in SGDBM

In GDBM, a data file (database) on disk is made up of a number of data structures including one file header, one hash directory, a number of hash buckets, a couple of available stacks, and the data blocks to hold real key / data pairs. Any manipulation of the data in a database must relate to one or several of these data structures. Each kind of original data structure and the related modification to make it secure is going to be presented below.

4.4.1.1 File Header

In GDBM, each database has one and only one file header. Moreover, the file header must be the first block of the whole data file. The file header keeps track of the current location of the hash directory and the unused free space in a data file. Each data file has a parameter to determine each block's size what is called *block_size*. It is set to 512 bytes by default and can be designated by users when creating a new data file. The file header structure must occupy exactly one block in size and always resides in the system memory. Whenever the location or content of the hash directory or any hash bucket changes, the file header may need to be updated to make sure all the information of the database is up to date. In GDBM, the original data structure for a file header is:

Original Structure:

```
typedef struct {
   int   header_magic;     /* 0x13579ace to make sure the header is good. */
   int   block_size;       /* The  optimal i/o block size from stat. */
   off_t dir;              /* File address of hash directory table. */
   int   dir_size;         /* Size in bytes of the whole directory table.  */
   int   dir_bits;         /* The number of address bits used to locate in the table.*/
   int   bucket_size;      /* Size in bytes of a whole hash bucket. */
   int   bucket_elems;     /* Number of elements in each  hash bucket. */
   off_t next_block;       /* The file address of the AVAIL_STACK if available. */
   avail_block avail;      /* This must be last because of the pseudo array in avail.
                              This avail grows to fill the entire block. */
} gdbm_file_header;
```

The Blowfish always requires that the data to be encrypted or decrypted is a multiple of 64-bit (8 bytes) in length. The default size of a block (512 bytes) is exactly a multiple of 64-bit so that the Blowfish can process the data of one or several block size directly and correctly. If a user wants to create a new database with his own block size, he must make sure that the block size is exactly a multiple of 64-bit. If the block size given by the user is less than 512 bytes, the system will ignore it and take its default block size instead.

Because a file header always takes one block and the block size is restricted to be a multiple of 64-bit, the size of a file header structure needs to be kept one block at any time. Both before encryption and after decryption, the hash value of the whole file header needs to be calculated by SHA-1 to check if the content of the file header has been tampered. This hash value must be stored together with the content of the file header for future comparison. The hash value (message digest) of SHA-1 is always 160 bits (20 bytes) in size. It is not a multiple of 64-bit. However, it is unnecessary to worry that its length will conflict with the Blowfish's requirement, because the last member variable of a file header structure is variable in size.

In the implementation of SGDBM, a new member variable was added into the original file header structure *gdbm_file_header*.

```
unsigned long small_hash[5];
```

This variable is used to hold the hash value of only the first two member variables of the structure *gdbm_file_header*. When opening an existing database, the whole file header is always put into the system memory first for future use. The file header takes one block, so a one-block memory space needs to be allocated. However, the block size itself is still unknown. Therefore, the data related to the block size has to be pulled out at the beginning. In SGDBM, even for such a small piece of data, it is also necessary to make it secure all the time. Therefore, the above number variable was added to do so. In addition, a new data structure type *header* was created. It contains only two member variables. One is an object of the modified *gdbm_file_header* structure and the other holds the hash value for this object. The new structure *header* takes one block and is sure to occupy the first block of the whole data file all the time. The following is the modified structure for a file header:

Modified Structure:

```
 typedef struct {
int   header_magic;     /* 0x13579ace to make sure the header is good. */
int   block_size;          /* The  optimal i/o block size from stat. */
     unsigned long small_hash[5];     /* A new variable used for header_magic and block_size. */
off_t dir;                          /* File address of hash directory table. */
int   dir_size;                     /* Size in bytes of the whole directory table.  */
int   dir_bits;                     /* The number of address bits used to locate in the table.*/
int   bucket_size;          /* Size in bytes of a whole hash bucket. */
int   bucket_elems;       /* Number of elements in each  hash bucket. */
off_t next_block;          /* The file address of the AVAIL_STACK if available. */
avail_block avail;        /* This must be last because of the pseudo array in avail.*/
 } gdbm_file_header;

 typedef struct {
unsigned long  hash_val[5];      /* The hash value for the whole gdbm_file_header. */
gdbm_file_header   gdbm_header;
```

```
    } header;                    /* A new data structure added in SGDBM */
```

4.4.1.2 Hash Directory

The hash directory itself is an extendible hash table. It is used to store file addresses of all hash buckets in a database. Any hash bucket can be located with its file address stored in the hash directory. In the original GDBM, when creating a new database, the initialized size of the hash directory is one block. Moreover, it initially takes the second block of the data file. With the subsequent database use, the size of the hash directory will keep increasing. In any case, it must be maintained to be a multiple of *block_size* in size. There is no such specific data structure designed for the hash directory. How to deal with the hash directory in SGDBM will be presented in section 3.4.2.2.1.

4.4.1.3 Hash Buckets

A hash directory is used to locate all hash buckets in a database. A hash bucket itself is a small hash table. Each hash bucket consists of a number of bucket elements plus some bookkeeping fields. A bucket element is used to hold detailed information corresponding to a specific key / data pair. The total number of bucket elements in a hash bucket depends on the optimum block size for the storage device and on a parameter given at the file creation time.

When a hash bucket gets full, it will be split into two hash buckets. The contents are split between them by the use of the first few bits of the 31-bit hash value of the corresponding key / data pair. The location of each bucket element in a hash bucket is calculated by the

31-bit hash value modulo the size of the bucket. To speed up writing, each hash bucket has an array with fixed amount of available elements. Each available element *avail_elem* holds the size and file address of an available space in the data file that can be used to store some key / data pairs. In GDBM, a hash bucket uses one block. The original data structure for a hash bucket is as follows:

Original Structure:

```
typedef struct {
     int   av_count;                        /* The number of  bucket_avail entries. */
     avail_elem  bucket_avail[BUCKET_AVAIL];    /* The avail_elems  for this bucket.*/
    int   bucket_bits;                      /* The number of bits used to locate bucket_elems in this bucket. */
    int   count;                            /* The real number of BUCKET_ELEMs in this bucket. */
    bucket_element h_table[1];        /*The BUCKET_ELEM table.Make it look like an array.*/
} hash_bucket;
```

In SGDBM, a new data structure type *bucket* was created. It contains two member variables. One is an object of the structure *hash_bucket* and the other is an array holding a 160-bit hash value of this object. The new structure *bucket* should fit in exactly one block. The following is the modified structure for the hash bucket:

Modified Structure:

```
typedef struct {
    unsigned long hash_val[5];       /* The hash value for the whole hash_bucket. */
    hash_bucket   bucket;
} bucket;                                /* A new data structure added in SGDBM */
```

A hash bucket contains a fixed amount of bucket elements. A bucket element can be located in the hash bucket by the 31-bit hash value of the corresponding key / data pair modulo the size of the hash bucket. A bucket element is used to hold the necessary detailed information of a specific key / data pair. The key / data pair itself is stored in an available element but not a bucket element. Each bucket element corresponds to one specific key / data pair. It contains the full 31-bit hash value and the file address of the corresponding key / data pair with both key size and data size. It also includes a small part of the actual key value what is used to verify the first part of the key if it has the correct value without having to read the whole key when retrieving a specific key / data pair. In GDBM, the original data structure for a bucket element is:

Original Structure:

```
typedef struct {
    int   hash_value;          /* The complete 31 bit hash value of the key/data. */
    char  key_start[SMALL];    /* Up to the first SMALL bytes of the key.  */
    off_t data_pointer;        /* The file address of the key. The data directly follows the key.  */
    int   key_size;            /* Size of key in key/data pair in the file. */
    int   data_size;           /* Size of associated data in key/data pair in the file. */
} bucket_element;
```

In SGDBM, two member variables were added into the original bucket element structure *bucket_element*. The first variable added is:

```
int   pad_size;
```

Originally, a key / data pair itself might be any length. In SGDBM, to meet the Blowfish size requirement, the size of a key / data pair must be a multiple of 64-bit. Therefore, it is necessary to pad any key / data pair to make sure that the whole size including the key, the data and the padding meets the size requirement. The size of the padding for a key / data pair is stored in the bucket element corresponding to this key / data pair so that the key / data can be picked up correctly.

The other member variable added is:

unsigned long hash_val[5];

This is the 160-bit hash value of each padded key / data pair used for future comparison to check if the key / data is tampered by some unauthorized person. The following is the modified data structure for a bucket element:

Modified Structue:

```
 typedef struct {
int   hash_value;                  /* The complete 31 bit hash value of the key/data. */
char  key_start[SMALL];       /* Up to the first SMALL bytes of the key.  */
off_t data_pointer;          /* The file address of the key. The data directly follows the key.  */
int   key_size;                    /* Size of key in key/data pair in the file. */
int   data_size;                   /* Size of associated data in key/data pair in the file. */
int   pad_size;                  /* A new variable used to store the size of padding */
    unsigned long  hash_val[5];     /* A new variable used to store the 160-bit hash value of the
                                              padded key/data pair */
 } bucket_element;
```

A bucket element is always read or written within the hash bucket that contains it and is never read from or written to the data file on disk separately. Therefore, it is not required to meet the Blowfish size requirement.

4.4.1.4 Available Tables

As mentioned in the last section, an available element *avail_elem* holds the size and file address of an available space in the data file that can be used to store some key / data pairs. All available elements in a database are organized in a number of available tables. An available table contains a number of available elements and sometimes bookkeeping fields. The most frequently used available table resides within the unique file header of a data file. Another available table that are also called available stack can be anywhere in a data file and in any size less than one block size. Most available tables reside, in fact, in the hash buckets in a database as available element arrays without bookkeeping fields.

When the available table in a file header fills up, it will be split and one half of the available elements will be pushed into a unique available stack located by the field *next_block* in the file header. When the available table in a file header is empty and the available stack is not empty, the top available elements of the available stack will be popped into the available table in the file header. The original data structure for the unique available stack in a database is:

Original Structure:

```
typedef struct {
    int   size;                    /* The number of avail elements in the available table.*/
   int   count;              /* The number of entries in the available table. */
   off_t next_block;              /* The file address of next unused disk block. */
   avail_elem av_table[1];     /* The avail table. Make it look like an array.  */
    } avail_block;
```

In reality, there is only one place to read and write the available table structure. It is for the use of the available stack. All other available tables either in the file header or in any hash bucket are never read from or written to the data file on disk separately. In SGDBM, a new data structure type *avail_stack* was created. It contains two member variables. One variable is an object of the original available stack structure *avail_block* and the other holds a 160-bit hash value of this object. The new structure *avail_stack* must take less than one block in size. The following is the modified structure for an available stack:

Modified Structure:

```
    typedef struct {
   unsigned long hash_val[5];       /* The hash value for the whole avail_block. */
   avail_block   stack;
    } avail_stack;                         /* A new data structure added in SGDBM */
```

An available element is used to store the information of an available space in a data file. It might appear in three kinds of data structures: the available table in the file header, the unique available stack and available element arrays in hash buckets. In GDBM, the data structure for an available element is:

Original Structure:

```
typedef struct {
    int   av_size;                  /* The size of the available space for storage. */
    off_t  av_adr;                  /* The file address of this available space in the data file. */
} avail_elem;
```

The size of an available element is not required to meet the Blowfish size requirement, because an available element is never read from or written to a data file separately. It is always read or written within an available table or a hash bucket that containing it. Therefore, there is no need to modify this data structure. It is kept in the original form.

All the data structures mentioned so far are the components that really exist in a data file on disk. That is to say, a data file is always made up of one or a number of each of these data structures presented above. Other than these data structures, for the purpose of convenience and performance, the GDBM has some other data structures that are only used in the system memory. They are presented in the following sections.

4.4.1.5 Key / Data Pairs

This data structure is used by both key and data of each key / data pair for compatibility. It contains the size and memory address of a specific key or data. It is used only in the system memory. No modification is made to this data structure.

Original Structure:

```
typedef struct {
    char * dptr;        /* The memory address of the data */
    int   dsize;         /* The size of the data */
 } datum;
```

4.4.1.6 Bucket Data in Cache

In order to avoid reading hash buckets from the data file on disk as much as possible, the GDBM gives two other data structures implemented in the system memory. One structure is used to hold cache elements. The other is used to hold the data in those cache elements. When initializing a database, the system will create a hash bucket cache that contains 100 cache elements. The hash bucket cache resides in another data structure designed for the whole database that is presented later. Each cache element in the hash bucket cache in the system memory corresponds to a specific real hash bucket in a data file on disk. When the hash bucket cache gets full, some hash buckets will be dropped in the least recently reading from disk order. In GDBM, the original data structure for a cache element is:

Original Structure:

```
 typedef struct {
     hash_bucket  *  ca_bucket;       /* Point to a real hash bucket read into the memory */
    off_t   ca_adr;                    /* The file address of corresponding hash bucket on disk */
    char  ca_changed;                  /* Indicate if some data in corresponding hash bucket changed. */
    data_cache_elem  ca_data;
 } cache_elem;
```

In SGDBM, the *ca_bucket* is make to point to not the original data structure *hash_bucket* but the new created data structure *bucket* presented in the section 3.4.1.3 so that it can contain the information of SHA-1 for future use. The modified data structure *cache_elem* is:

Modified Structure:

```
  typedef struct {
  bucket  *  ca_bucket;           /* Point to a real bucket read into the memory */
 off_t   ca_adr;                  /* The file address of corresponding hash bucket on disk */
 char  ca_changed;                /* Indicate if some data in corresponding hash bucket changed. */
 data_cache_elem  ca_data;
  } cache_elem;
```

To speed up fetching and "sequential" access, it is necessary to define a data structure to catch the information of the key / data pair in the hash bucket corresponding to a specific cache element in the system memory. Data structure *data_cache_elem* is designed for this purpose. Each hash bucket corresponding to a specific *cache_elem* has only one *data_cache_elem* to store the information for a specific key / data pair in this hash bucket.

To find a specific key in a database, usually it has to match exactly this key with all keys in the data file on disk. To reduce overhead, the data associated with each key will be read into memory together at the same time. Any key / data pair read from the data file into memory is stored together at the memory address indicated in the field *dptr* of the data structure *data_cache_elem*. In GDBM, the original data structure for a key / data pair cache element is:

Original Structure:

```
typedef struct {
     int   hash_val;       /* The complete 3- bit hash value of the key/data pair. */
    int   data_size;
    int   key_size;
    char  *  dptr;         /* Memory address to store the real key / data  pair.*/
    int   elem_loc;       /* The location of this key/data in the corresponding hash bucket.*/
} data_cache_elem;
```

In SGDBM, a new member variable used for padding size was added into the original data structure *data_cache_elem*. The original data structure *bucket_elem* was changed in the former section 3.4.1.3, so it is necessary to make the corresponding modification herein for consistence. The modified data structure *data_cache_elem* is:

Modified Structure:

```
typedef struct {
    int   hash_val;        /* The complete 3- bit hash value of the key/data pair. */
    int   data_size;
    int   key_size;
    int   pad_size;         /* a new variable to store the size of padding for key/data pairs */
    char  *  dptr;          /* Memory address to store the real key / data  pair.*/
    int   elem_loc;        /* The location of this key/data in the corresponding hash bucket.*/
} data_cache_elem;
```

4.4.1.7 GDBM Data File Information

When opening a database, the system will allocate some space in the system memory to hold the overall information of the database. In GDBM, it defines a special data structure *gdbm_file_info* that contains all memory-based information for a database. It allows multiple databases to be opened at the same time by creating multiple objects of this data structure. This data structure is used only in the system memory for various databases manipulations. In GDBM, the original data structure for a data file is:

Original Structure:

```
typedef struct {
   /* Global variables and pointers to dynamic variables used by gdbm.*/
   char * name;                     /* The data file name. */
   int  read_write;                /* The reader / writer status. */
   int  fast_write;                 /* It is set to 1 if no fsyncs is to be done.*/
   int  central_free;               /* Set to 1 if all free blocks are kept in file header.*/
   int  coalesce_blocks;            /* set to 1 if we should try to merge free blocks.*/
   int  file_locking;               /* If we should do file locking ourselves. */
   void ( * fatal_err) ( );         /* The fatal error handling routine. */
   int  desc;                       /* The data file descriptor set by function gdbm_open().*/
   gdbm_file_header * header;       /* Point to the file header of the database */
   off_t * dir;                     /* Point to the hash directory of the database. */
   cache_elem * bucket_cache;       /* A hash  bucket cache with cache_size bucket cache elements.*/
   int  cache_size;                 /* Here is 100. */
   int  last_read;                  /* Indicate the last read cache element. */
   hash_bucket * bucket             /* Points to the current hash bucket in the cache.*/
   int  bucket_dir;                 /* The directory entry for the current hash bucket.*/
```

```
    cache_elem * cache_entry;       /* Point to the current hash bucket's cache entry. */
char  header_changed;             /* Things need to be written back to disk when updated.*/
char  directory_changed;
char  bucket_changed;
char  second_changed;
 } gdbm_file_info;
```

In SGDBM, two lines of the original data structure *gdbm_file_info* are modified. First, the pointer *header* is made pointing to the file header of a data file, which points to the modified data structure *header* but not the original data structure *gdbm_file_header*. Then the pointer *bucket* is made pointing to the current hash bucket, which points to the modified data structure *bucket* but not the original data structure *hash_bucket*. The purpose of both modifications is to keep consistent with the prior modifications to relevant data structures for security sake. The modified data structure for a data file is:

Modified Structure:

```
 typedef struct {
char * name;                      /* The data file name. */
int  read_write;                  /* The reader / writer status. */
int  fast_write;                       /* It is set to 1 if no fsyncs is to be done.*/
int  central_free;                 /* Set to 1 if all free blocks are kept in file header.*/
int  coalesce_blocks;              /* set to 1 if we should try to merge free blocks.*/
int  file_locking;                 /* If we should do file locking ourselves. */
void ( * fatal_err) ( );           /* The fatal error handling routine. */
int  desc;                              /* The data file descriptor set by function gdbm_open().*/
     header * header;                   /* Changed to point to the modified file header structure */
```

```
    off_t * dir;                    /* Point to the hash directory of the database. */
    cache_elem * bucket_cache;      /* A hash  bucket cache with cache_size bucket cache elements.*/
    int  cache_size;                /* Here is 100. */
    int  last_read;                 /* Indicate the last read cache element. */
     bucket * bucket;               /* Changed to point to the modified hash bucket structure */
    int  bucket_dir;                /* The directory entry for the current hash bucket.*/
    cache_elem * cache_entry;       /* Point to the current hash bucket's cache entry. */
    char  header_changed;           /* Things need to be written back to disk when updated.*/
    char  directory_changed;
    char  bucket_changed;
    char  second_changed;
} gdbm_file_info;
```

4.4.2 Database Functions in SGDBM

In the previous sections, all of the data structures in SGDBM were presented. It has been mentioned that a database itself consists of these data structures completely. In fact, all kinds of database manipulations, no matter of users' data or metadata in a database, must relate to one or several of these data structures. In the following sections, some functions provided by GDBM are going to be presented first, and then the implementation of all functions in SGDBM will be presented.

4.4.2.1 Functions in GDBM

In GDBM, there are two major types of functions. One type is external functions provided as interface to extend programs. All of the external functions in GDBM are listed in the header file *gdbm.h* and can be called by any program. The other type of functions is internal functions used by external functions or other GDBM internal functions. The implementation of SGDBM is tightly based on all of these internal and external functions in GDBM. Therefore it is necessary to present these functions in the first place. In the following sections, the external and internal functions in GDBM will be introduced respectively.

4.4.2.1.1 External Functions in GDBM

Almost all the external functions in GDBM are listed with short descriptions below. Users can use these external functions by including the header file *gdbm.h* in their programs.

Function:

*gdbm_file_info * gdbm_open (char * name, int block_size, int flags, int mode, void(* fatal_func)());*

This function is used to create a new database or to open an existing database. When the name of the data file designated by the parameter *name* does not exist, it will create a new database. The parameter *block_size* is a basic unit of data file I/O operations. It is also used to determine the size of various data structures in GDBM. If the given value is less than 512, the file system block size is used. The parameter *flags* has several options like GDBM_READER, GDBM_WRITER, GDBM_WRCREAT and GDBM_NEWDB. It decides how user can access a database. If no error occurs, a pointer to the "gdbm file descriptor" will be returned. This descriptor will be used later by all other GDBM functions to access this opened database.

Function:

*void gdbm_close (gdbm_file_info * dbf);*

Whenever a data file is not needed any longer, it must be closed. It is required to update the reader / writer count on a specific data file. This function will close a database pointed by the given descriptor and free all memory spaces associated with the database.

Function:

*int gdbm_store (gdbm_file_info * dbf, datum key, datum content, int flag);*

This function is used to actually store key / data pairs into a database. The parameter *flag* is used to decide what happens if the given key exists already. If its value is GDBM_REPLACE, the old data associated with the given key is replaced by the new content. If its value is GDBM_INSERT, an error will be returned and no action is taken if the key already exists. The size of both keys and data is not restricted. This function returns 0 when it executes successfully. Otherwise, it returns 1 or –1.

Function:

*datum gdbm_fetch (gdbm_file_info * dbf, datum key);*

*int gdbm_exists (gdbm_file_info * dbf, datum key);*

These two functions are used to fetch key / data pairs from a database. The first function returns the data associated with the given key if no error takes place. The second function is used to search for a particular key without retrieving the entire key / data pair. Unlike the first function, it does not allocate any memory and simply returns a value of true or false, depending on whether the given key exists or not.

Function:

*int gdbm_delete (gdbm_file_info * dbf, datum key);*

This function is used to remove the entire key / data pair associated with the given key from a database. It returns 0 when successful and –1 otherwise.

Function:

*datum gdbm_firstkey (gdbm_file_info * dbf);*

*datum gdbm_nextkey (gdbm_file_info * dbf, datum key);*

These functions allow for accessing all key / data pairs in a database. The access order is not sequential and has to do with the 31-bit hash values of key / data pairs. The first function starts the visit of all keys in the database. The other function finds and reads the next entry in the hash directory of the database. They both return the data associated with a specific key.

Function:

*int gdbm_reorganize (gdbm_file_info * dbf);*

This function is used to reorganize the whole database by creating a new data file and inserting all key / data pairs in the old data file into the new data file. This function can shorten the size of the data file after a large number of deletions.

Function:

*void gdbm_sync (gdbm_file_info * dbf);*

This function allows the programmer to make sure the disk version of a database has been completely updated with all changes to the time of execution of this function. This function is usually called after a complete set of changes has been made to the database, to record these changes permanently.

Function:

*int gdbm_setopt (gdbm_file_info * dbf, int option, int * value, int size)*

This function is used to set certain options on an already opened database. The valid options include GDBM_CACHESIZE and GDBM_FASTMODE.

4.4.2.1.2 Internal Functions in GDBM

As previously mentioned, the external functions provided by GDBM are interface for users. The users seldom directly manipulate the data structures of a database like what is mentioned in the previous section. They usually invoke some functions (API) that can actually manipulate various metadata in a database. These functions are called internal functions and implement the external functions in GDBM. These functions are organized into some source files according to the type of the data structures they handle. These internal functions are presented briefly below:

Function:

(From bucket.c)

*void _gdbm_new_bucket (gdbm_file_info * dbf, hash_bucket * bucket, int bits)*

*void _gdbm_get_bucket (gdbm_file_info * dbf, int dir_index)*

*void _gdbm_split_bucket (gdbm_file_info * dbf, int next_insert)*

*void _gdbm_write_bucket (gdbm_file_info * dbf, cache_elem * ca_entry)*

These functions are specially used to process hash buckets in a database. The first function is used to initialize a new hash bucket and set hash values of all bucket elements in the new hash bucket to –1 to indicate that these bucket elements are not yet used and could be used by any key /data pair in the future. The second function is used to find a hash bucket in a database that is pointed by the hash directory from the location *dir_index*. The third function is used to split the current hash bucket into two new hash buckets. The last function is used to actually write the hash bucket pointed by the given cache element in system memory to the data file. This is the only place where a hash bucket is written to a data file on disk.

Function:

(From falloc.c)

```
off_t _gdbm_alloc ( gdbm_file_info * dbf, int num_bytes )
void _gdbm_free ( gdbm_file_info * dbf, off_t file_adr, int num_bytes )
int _gdbm_put_av_elem ( avail_elem new_el, avail_elem av_table[], int *av_count, int can_merge )
static void push_avail_block ( gdbm_file_info * dbf )
static void pop_avail_block ( gdbm_file_info * dbf )
static void adjust_bucket_avail ( gdbm_file_info * dbf )
static avail_elem get_elem ( int size, avail_elem av_table[], int * av_count )
static avail_elem get_block ( int size, gdbm_file_info * dbf )
```

These functions are specially used to process available blocks and available elements in a database. As mentioned before, an available element *avail_elem* is used to hold the size and file address of an available space in a data file. An available block is an actual available disk space that can store key /data pairs themselves but not their corresponding information. In other words, each available element holds the information of a specific available block. The first three functions are usually called by other external functions. The last five functions are just invoked by the first three functions.

The first function is used to find an available block in a data file with the size *of at least num_bytes*. It returns the file address of the start point of the available block. The second function is used to free a piece of disk space at the file address *file_adr* in the data file with the size of *num_bytes* for reuse. This freed disk space will be treated as a new available block and the corresponding available element will be added to a specific available table. The third function is used to insert an available element into a specific available table with possible merge.

The fourth function is used to push a half of the available elements in the available table in a file header into the unique available stack when the available table in the file header is full. The fifth function is used to pop a half of the available elements in the unique available stack to the available table in a file header when the available table in the file header is less than half full. The sixth function is used to balance the amount of available elements between the available table in a file header and the available stack in the data file. The seventh function is used to find an available element with the size not less then *size* from the given available table and returns this available element. The last function is used to locate a never used space with the size of *size* from the end of the data file and returns a new available element corresponding to this new available block.

Function:

(From findkey.c)

*char * _gdbm_read_entry (gdbm_file_info * dbf, int elem_loc)*

*int _gdbm_findkey (gdbm_file_info * dbf, datum key, char ** dptr, int * new_hash_val)*

These functions are used to handle real key / data pairs. The first function is used to obtain the key / data pair found in the given bucket entry *elem_loc* in the current hash bucket. It will cache the real key / data pair in the system memory and return the pointer to this pair. The second function returns the bucket entry for the given key in the current hash bucket and returns a pointer *dptr* to the key / data pair in the system memory if it is found.

Function:

(From update.c)

*void _gdbm_end_update (gdbm_file_info * dbf)*

*static void writeheader (gdbm_file_info * dbf)*

The first function is used to write all the changed metadata in the system memory including changed file header, changed hash directory and any changed hash bucket back to the data file on disk. It will invoke the second function, which is used to write the changed file header in the system memory back to the data file on disk.

Function:

(From gdbmopen.c)

*int _gdbm_init_cache (gdbm_file_info * dbf, int size)*

This function is used to initialize a hash bucket cache in the system memory with *size* of cache elements of the data structure *cache_elem* when opening a database.

Function:

(From hash.c)

int _gdbm_hash (datum key)

This function is used to compute the 31-bit hash value for each key / data pair. This hash value is used in two places. The top n-bit is used to locate in a hash directory to find in which hash bucket hold the bucket element corresponding to this key / data pair. The other use is to determine the location *elem_loc* of the bucket element that hold the information of this key / data pair in the corresponding hash bucket. This location is obtained by module this 31-bit hash value by the size of the hash bucket.

4.4.2.2 Functions in SGDBM

After analyzing both original data structures and functions in GDBM, all things need to be done to transform the GDBM to be the SGDBM are clear. All the data structures necessary to be modified were present already in the previous sections. In this section, it is going to describe how to modify the related functions.

In GDBM, only some of its external and internal functions directly manipulate the data structures of a database. These functions are exactly things necessary to be modified to transfer the GDBM to be the SGDBM. They are given below together with their associated data structures:

All data structures need to be read from a data file in GDBM and all associated functions that do these reading operations:

The first part of a file header: *partial-header*

*gdbm_file_info * gdbm_open (char * name, int block_size, int flags, int mode,* void(* fatal_func)());

(From gdbmopen.c)

The entire file header: *header*

*gdbm_file_info * gdbm_open (char * name, int block_size, int flags, int mode,* void(* fatal_func)());

(From gdbmopen.c)

The entire hash directory: *dir*

*gdbm_file_info * gdbm_open (char * name, int block_size, int flags, int mode, void(* fatal_func)());*

(From gdbmopen.c)

The entire hash bucket: *bucket*

*void _gdbm_get_bucket (gdbm_file_info * dbf, int dir_index)*

(From bucket.c)

The entire available stack: *avail_stack*

*static void pop_avail_block (gdbm_file_info * dbf)*

(From falloc.c)

The entire key / data pair: *datum*

*char * _gdbm_read_entry (gdbm_file_info * dbf, int elem_loc)*

(From findkey.c)

All data structures need to be written to a data file in GDBM and all associated functions that do these writing operations:

The entire file header: *header*

*gdbm_file_info * gdbm_open (char * name, int block_size, int flags, int mode, void(* fatal_func)());*

(From gdbmopen.c)

*static void writeheader (gdbm_file_info * dbf)*

(From update.c)

The entire hash directory: *dir*

*gdbm_file_info * gdbm_open (char * name, int block_size, int flags, int mode, void(* fatal_func)());*

(From gdbmopen.c)

*void _gdbm_end_update (gdbm_file_info * dbf)*

(From update.c)

The entire hash bucket: *bucket*

*gdbm_file_info * gdbm_open (char * name, int block_size, int flags, int mode, void(* fatal_func)());*

(From gdbmopen.c)

*void _gdbm_write_bucket (gdbm_file_info * dbf, cache_elem * ca_entry)*

(From bucket.c)

The entire available stack: *avail_stack*

*static void push_avail_block (gdbm_file_info * dbf)*

(From falloc.c)

The key of a key / data pair: *datum*

*int gdbm_store (gdbm_file_info * dbf, datum key, datum content, int flag);*

(From gdbmstore.c)

The data of a key / data pair: *datum*

*int gdbm_store (gdbm_file_info * dbf, datum key, datum content, int flag);*

(From gdbmstore.c)

Other than the functions listed above, all other functions in GDBM do not involve in disk I/O operations. That is to say, those functions will never read from or write to a data file on disk directly. They just deal with some kinds of data structures in the system memory. How to modify the functions listed above to implement security object will be presented in the section 4.4.2.2.2.

4.4.2.2.1 Secure Functions

Security requires alterations to two respects of implementation. One is encryption and decryption by Blowfish. It can keep data from being read by unauthorized users. The other is detection, using SHA-1, to determine if a data file has been tampered. Regardless of the data structures mentioned above, all the functions that directly manipulate the data structures of a database on disk need to implement these two operations for security. A new *C* file *readwrite.c* was created with a number of functions that focus on these two security objects and will be invoked by some of the functions that manipulate data structures of a database directly. These secure functions are as follows:

Function:

```
char *  GetUserPwdKey();
char *  GetHarddiskKey();
char *  GetPSNKey();
char * gdbm_GetBlowfishKey( );
```

These functions are used to generate keys for the Blowfish encryption or decryption in SGDBM. The components of these keys have been presented in detail in the section 4.1.1. A key consists of three parts: the encrypted user’s login password, the serial number of the first IDE hard disk and the processor serial number (PSN) of the CPU currently used. The first three functions are implemented to get these three parts of the key respectively. The last function invokes first three functions to generate the entire key for Blowfish.

Function:

void gdbm_HashError( );

This function is used to indicate an error whenever the 160-bit hash value of some data recalculated by SHA-1 is different with the original hash value.

Function:

*int gdbm_Read_Metadata (void * metadata, unsigned long * buffer, int metadata_size);*

*int gdbm_Read_Dir (unsigned long * dir, unsigned long * buffer, int dir_size);*

These two functions are used, for the purpose of security, to handle the data structures of metadata after they are read from a data file on disk. The first function is used to handle most of the data structures of metadata including the file header, the hash bucket and the available stack. In this function, it first decrypts the encrypted metadata just read from a data file into the *buffer* in the system memory. Then it recalculates a 160-bit hash value of the metadata calling related SHA-1 functions and compares it with the original hash value saved in the metadata data structure to see if the metadata has been tampered. If the hash value is not same as before, an error will be raised.

The second function is specialized for the hash directory and does the same thing as the first function. The reason to separate it from the other kinds of metadata is due to the size differences between these data structures. Other than the hash directory, any data structure of the metadata that will be read from a data file takes one and only one *block_size* all the time. Therefore, they can be processed in the same way. However, the size of a hash directory keeps increasing with the advance of time. At the time that a database is created, the hash directory is initialized in just one block. With the occurrences of more and more hash buckets, the hash directory must be increased accordingly to be able to index all hash buckets. In any case, it must be a multiple of *block_size* plus 24 bytes used to hold a 160-bit hash value for the whole hash directory and four padding bytes to meet the Blowfish size requirement.

Function:

*int gdbm_Write_Metadata (void * metadata, void * buffer, int metadata_size);*

*int gdbm_Write_Dir (unsigned long * dir, void * buffer, int dir_size, int block_size);*

These two functions are used to process data structures of metadata for the purpose of security before they are written to a data file on disk. The first function is used to handle most of the data structures of metadata including the file header, the hash bucket and the available stack. In this function, it first calls relevant SHA-1 functions to calculate a 160-bit hash value of the metadata and save it into the relevant data structure as a member variable. Then, the entire data structure containing the 160-bit hash value is encrypted by Blowfish and the encrypted data structure will be put in the given *buffer* in the system memory.

The second function is specialized for the hash directory and does the same thing as the first function. The reason to separate it from the other kinds of metadata is also due to the size differences between these data structures. The reason is the same as the one for two functions used to read metadata presented right above.

4.4.2.2.2 Functions Modified for Purpose of Security

After introducing above functions that focus on security, let us return to the functions that directly manipulate data structures of a database on disk. For each kind of data structure, the implementation of security parts within the associated functions will be presented in detail below.

File Header:

Whenever opening an existing database in SGDBM, the unique file header of the database will be read into the system memory from the data file on disk only once. Once the entire file header is read into the system memory, some of its fields may change owning to various future database operations. Therefore, the updated file header needs to be written back to the data file on disk.

As previously mentioned, in GDBM, a file header always takes one and only one block. However, the value of *block_size* is not known yet because *block_size* is also a field of the file header. Without this *block_size*, the system is not able to allocate appropriate memory space for the file header. The solution is to read the first partial data of the file header at the beginning of opening an existing database. According to the data structures for a file header in SGDBM:

```
typedef struct {
   int   header_magic;
   int   block_size;
    unsigned long small_hash[5];     /* 160-bit hash value of header_magic and block_size fields */
   ………
} gdbm_file_header;


typedef struct {
unsigned long  hash_val[5];     /* 160-bit hash value of the gdbm_header object */
gdbm_file_header   gdbm_header;
} header;
```

At the beginning of opening an existing database, the system first reads the part of the file header *header* containing the member variable *hash_val[5]* and first three member variables of the data structure *gdbm_file_header* from the beginning of the data file into the system memory. The size of this part of the file header is fixed so that the system knows how much memory space should be allocated. This fixed size is two integers plus two unsigned long integers. Then it decrypts this part of file header and calculates a 160-bit hash value of the first two fields of the *gdbm_file_header*: *magic and block_size*. It compares the new hash value with the old one stored in the member variable *small_hash[5]* to see whether this part of the file header has been tampered. If they are the same, the system uses the decrypted value of *block_size* to allocate memory space for the entire file header *header*. Finally, it reads the entire file header in the first block of a data file into the system memory and invokes the function *gdbm_Read_Metadata( )* to decrypt the file header *header* and verify the hash value. All of above operations related to reading the file header of an existing database are implemented in the function:

```
gdbm_file_info * gdbm_open ( char * name, int block_size, int flags, int mode, void(*  fatal_func)() );
```

As for writing a file header to a data file on disk, it involves two functions. The content of a file header in the system memory is often changed owing to various database operations and needs to be written back to the corresponding data file on disk at the time of updating metadata. The function:

*static void writeheader (gdbm_file_info * dbf)*

invokes the function *gdbm_Write_Metadata( )* to do the encryption and 160-bit hash value calculation. Then this function writes the encrypted and updated file header back to the data file on disk.

The other place involved in writing a file header is the function:

*gdbm_file_info * gdbm_open (char * name, int block_size, int flags, int mode, void(* fatal_func)());*

In this function, when trying to open a data file not existing on disk, it will create a new data file with the given file name and initialize some metadata to generate the new database. First, it allocates some memory space for a file header *header*. Then it will initialize some fields of the file header. The field *header_magic* is set to a fixed value and used to check if the database is good when the database is opened in the future. The field *block_size* is set by the input parameter *block_size*. Afterwards, it calculates a 160-bit hash value of two fields *header_magic* and *block_size* and then stores this hash value in the field *small_hash[5]*. After initializing the available table of the file header and the first bucket in the data file, it invokes the function *gdbm_Write_Metadata( )* to process the initialized file header and finally writes the encrypted file header to the corresponding new data file on disk as the first block.

Hash Directory:

In SGDBM, the hash directory in a database is read into the system memory from the data file on disk only once whenever opening an existing database. In the function:

*gdbm_file_info * gdbm_open (char * name, int block_size, int flags, int mode, void(* fatal_func)());*

After reading the file header from the data file on disk, it will read the entire hash directory in the system memory. The file address of the hash directory is stored in the field *dir* of the file header. The size of the hash directory must always be a multiple of *block_size* plus 24 bytes. It allocates the memory space for the hash directory and reads the encrypted hash directory into memory. Then, it invokes the function *gdbm_Read_Dir( )* to decrypt the hash directory and verify the hash value to see if the hash directory is valid. As previously mentioned, in the function *gdbm_Read_Dir( ),* it will do some padding job to make the size of data meet the Blowfish size requirement.

Whenever the hash directory is changed, it should be written back to the data file on disk in time to keep the metadata up-to-date. In the function:

*void _gdbm_end_update (gdbm_file_info * dbf)*

It invokes the function *gdbm_Write_Dir( )* to process the entire hash directory by adding some padding characters and then writes the encrypted and updated hash directory back to the data file on disk. The size of a hash directory and the position of it in a data file may change at any time.

When creating a new database, in the function:

*gdbm_file_info * gdbm_open (char * name, int block_size, int flags, int mode, void(* fatal_func)());*

After initializing the file header for the new database, it will initialize the hash directory. In the new database, the initialized size of a hash directory is always exactly one *block_size* plus 24 bytes. It right follows the file header and occupies the second block in the new data file. The hash directory, fter being initialized and processed by the function *gdbm_Write_Dir( )*, will be written back to the new data file on disk.

Hash Bucket:

In SGDBM, each data file contains one or many hash buckets. Every hash bucket can be addressed in the hash directory of the corresponding database. When a specific hash bucket is needed, the following function can locate it in the hash directory and obtain the related file address and then read the encrypted hash bucket from the found file address on disk into the buffer allocated in advance in the system memory.

*void _gdbm_get_bucket (gdbm_file_info * dbf, int dir_index)*

After, this function will invoke the function *gdbm_Read_Metadata( )* to make this hash bucket readable in plain text.

The content of hash buckets keep on changing much more frequently than the data structures of other metadata. Whenever a specific hash bucket is changed, the entire hash bucket should be written back to the data file on disk in time. In the function:

*void _gdbm_write_bucket (gdbm_file_info * dbf, cache_elem * ca_entry)*

It invokes the function *gdbm_Write_Metadata( )* to process the changed hash bucket and then writes the encrypted and updated hash bucket back to the data file on disk. The size of a hash bucket never changes, but its position in a data file may change.

When creating a new database, in the function:

*gdbm_file_info * gdbm_open (char * name, int block_size, int flags, int mode, void(* fatal_func)());*

After initializing the file header and the hash directory, the sytem will initialize the first hash bucket for the new database. The size of a hash bucket *bucket* is always exactly one *block_size*. This first and the only hash bucket in a new database takes the third block of the new data file on disk right following the file header and the hash directory of this new database. After being initialized and processed by the function *gdbm_Write_Metadata( )*, the hash bucket will be written back to the new data file.

Available Stack:

In SGDBM, there is one and only one available stack in a data file. Both the size and the position of the available stack may vary sometimes. The size of an available stack is up to one *block_size*. The file address of an available stack is stored in the field *next_block* of the file header. Before it pops some available elements from the available table in a file header and inserts them into the available stack, it needs to read the entire available stack into the system memory from the data file on disk. The reading process is handled by the function:

*static void pop_avail_block (gdbm_file_info * dbf)*

Once the entire available stack is read into the memory, it will be decrypted and verified with the hash value by invoking the function *gdbm_Read_Metadata( )*.

After it pushes some available elements from the available stack into the available table in the file header, the content of the available stack must have been changed. Therefore, it is necessary to write the changed available stack back to the data file on disk. This process is implemented in the function:

*static void push_avail_block (gdbm_file_info * dbf)*

It invokes the function *gdbm_Write_Metadata( )* to process the changed available stack and then writes the encrypted and updated available stack back to the data file on disk.

Key / Data Pair:

In SGDBM, as mentioned before, the information of each key / data pair including its size and file address is stored in the corresponding bucket element *bucket_elem* of a hash bucket. A bucket element is read from or written to a data file within the hash bucket containing it. However, a key / data pair itself is stored separately with its information in a data file on disk. The process of reading or writing a key / data pair itself is different from the process of reading or writing its information. The key / data pairs may vary a lot in actual size, so it is necessary to consider adding some padding to meet the Blowfish size requirement. In addition, unlike those metadata that have the 160-bit hash value stored as a specific field of their data structures, the 160-bit hash value of a key / data pair is stored as a field of its corresponding bucket element, but not together with the key / data pair itself.

When reading a specific key / data pair from a data file on disk, the system first needs to get the size and file address of this key / data pair from its corresponding bucket element. Then it reads the entire encrypted key / data pair into the system memory from the found file address. This is implemented in the function:

*char * _gdbm_read_entry (gdbm_file_info * dbf, int elem_loc)*

Afterwards it will decrypt the entire key / data pair that actually contains the key, the data and the padding. It calculates the 160-bit hash value of only the key and the data but not the padding and then compares it with the old hash value stored in the relevant field of the corresponding bucket element. If the new hash value is the same as the old one, it can make sure that the key / data pair is valid and not tampered by unauthorized users.

When writing a specific key / data pair to a data file on disk, the writing process is implemented in the following function:

```
int gdbm_store ( gdbm_file_info * dbf, datum  key, datum  content, int flag );
```

The process of writing is a little more complex than reading. Because any key / data pair in a data file must meet the Blowfish size requirement, the system needs to make sure of the correct padding size for each key / data pair before it is written to a data file. Several different conditions need to be considered. If the key exists in the database already, it just replaces the data associated with the given key. The padding size depends on the new data size. If the new data is less then the old data in size, it still uses the original file address to store the key / data pair. Otherwise, it will free the original file space and allocate a new file address for this key / data pair. If the key does not exist in the database, that is to say, this is a new key / data pair in the database, it just calculates the padding size for this key / data pair to meet the Blowfish size requirement and then looks for a file address for this new key / data pair.

After getting the correct padding size, it calculates a 160-bit hash value of only the key and the data but not the padding. This hash value and the padding size will be stored in the relevant fields of the corresponding bucket element. Finally, it encrypts the key / data pair together with the padding and then writes the entire encrypted data to a data file at the file address stored in a specific field of the corresponding bucket element.

4.4.3 Other Implementation in SGDBM

In SGDBM, other than analyzing and modifying data structures and functions in the original GDBM, it is also necessary to modify a number of other files for the purpose of correct system compilation and installation. First, it is necessary to modify some original source code files, which define the internal and external functions, to include the new header files created in SGDBM. Second, in the header file *proto.h* that includes most of the prototypes for the GDBM routines, it is necessary to change the prototypes for those functions modified in SGDBM. Most importantly, the *Makefile.in* file, which is used to automatically generate the *Makefile* for the whole system, must be modified accordingly.

After all necessary files in the original GDBM were analyzed and modified, it becomes SGDBM, a totally secure database management system. All of the data in a database in SGDBM, regardless of user's data or metadata, is encrypted to prevent from reading by unauthorized users. In addition, using SHA-1, it always can detect whenever user's data or metadata in a database has been tampered. The system is not able to avoid tampering by other users on purpose, however, it can tell that the data is not valid any more. Before storing any data in a database, SGDBM calculate a hash value for the data and store the hash value together with the data. When fetching the data from the database, SGDBM recalculate the hash value for this data and compare it with the hash valued stored in the database. If they are same, it means the data has not been tampered. If not, the data might have already been tampered.

4.5 Implementation of CSFS

With the implementation of SGDBM, it can be used to store significant data that need to be protected. In the CINDI Secure File System (CSFS), the SGDBM is used to store file passwords. The implementation of CSFS will be presented in detail below.

In CSFS, regardless of the type of file, all files that are to be secure, are encrypted entirely to keep in secure status when stored on disk. When a user wants to save a file in CSFS, the system first calculates a 160-bit hash value of the file by SHA-1. Then it encrypts the entire file including this hash value by Blowfish. For encryption, the user needs to give the file a file password in any size up to 56 bytes. However, the user does not need to remember the file password because it is not necessary to be provided when this file is decrypted in the future. This way, the user can use any file password without making it easy to remember. Thus others cannot easily guess such file passwords to have files encrypted as in some simple file security policy.

In CSFS, a file password itself needs to be encrypted as well. It is encrypted by Blowfish with a specific key. This key is made up of three parts: the serial number of the first IDE hard disk, the processor serial number (PSN) of the CPU currently used and a user password directly inputted by the user. The encrypted file password will be stored in a SGDBM database for security. When decrypting a file, the user just needs to give the user password, which was inputted by him at the encryption time, as a part of the key to decrypt the file password for the required file. In this way, users have the flexibility to design their user passwords. Each user can either have only a single user password to encrypt all file passwords or use different user passwords for different file types. However, user passwords need to be given at the time of decrypting files, so users must always remember them.

An example is given here to facilitate understanding. A user Scott needs to use CSFS to make some of his files secure. He prefers to use different user passwords for different file types, such as ‘TigerTxt’ for all text files with the suffix ‘.txt’ and ‘TigerExe’ for all executable files with the suffix ‘.exe’, etc. When he wants to save a text file ‘Report.txt’, he first needs to input a file password, such as ‘AbCdEfG12345678’, for encrypting the text file itself. Then he needs to input a user password that is used as a part of the key to encrypt the above file password. Here it should be ‘TigerTxt’ because it is a text file. At the time of decrypting this text file, Scott just needs to input the user password ‘TigerTxt’ and it will be used as a part of the key to decrypt the file password stored in the SGDBM database. The file password is not necessary to be inputted at the decryption time. Therefore, Scott just needs to remember the user password ‘TigerTxt’, but not the file password ‘AbCdEfG12345678’. For another text file, Scott can input any other file password such as ‘9876543AddBcc$RE&’, but use the same user password ‘TigerTxt’, for any text file.

The system will calculate a 160-bit hash value for the key used to encrypt a file password by Blowfish. The file password followed by this hash value will be stored as the data of a key / data pair in a SGDBM database. The key of such a key / data pair is the file’s absolute path. That is to say, in CSFS, each file has one and only one corresponding key /

data pair in the SGDBM database. Taking the example given above, for the text file ‘Report.txt’, if it is saved in the directory ‘c:/Scott/’, the key of the key / data pair for this text file is ‘c:/Scott/Report.txt’. The file password ‘AbCdEfG12345678’ will be encrypted by Blowfish using the key containing the user password ‘TigerTxt’. Let us assume that the encrypted file password is ‘Ba#45d7&*3EG1f6’. The data of the key / data pair for this text file will be the encrypted file password ‘Ba#45d7&*3EG1f6’ plus the 160-bit hash value of the key containing the user password ‘TigerTxt’.

Such a key / data pair corresponding to a file in CSFS, will be processed by Blowfish and SHA-1 in SGDBM. Therefore, an unauthorized user can neither locate a specific file name along with its encrypted file password by reading the data file itself nor tamper any key / data pair without the user’s detection.

In addition, in CSFS, each user has his unique SGDBM database. The data file is created automatically, in the user’s home directory, at the first time the user saves a file in CSFS. Whenever the user is removed from the system, along with the removal of the user’s home directory, the corresponding SGDBM database will disappear. This way, the CSFS can guarantee that one user’s operation of his files will not affect another user’s files.

In CSFS, when a user wants to open a file, the system decrypts this file first. Before decrypting the file, the user password needs to be inputted by the user to generate the key used to decrypt the file password for this file. The key / data pair corresponding to this file is fetched from the SGDBM database. The system recalculates the 160-bit hash value of the key containing the user password and compares it with the old hash value, which is given at the encryption time as a part of the data of the key / data pair. If the new hash value is different from the old one, the system will issue a warning message that the user password is not correct and then stops the execution to keep the encrypted file on disk intact. If the new hash value is the same as the old one, the file password will be decrypted by Blowfish with the key containing the user password.

If the file password is correct, it will be used to decrypt the entire file that contains a 160-bit hash value of the original file. After decrypting, the system will recalculate a 160-bit hash value of the original file and compare it with the old hash value. If they are the same, it will open the file as its original type. If the newly calculated hash value is different from the previously stored hash value, the system issues a warning message that the file has been tampered and stops the execution. Therefore, the encrypted file on disk still keeps intact.

Taking the previous example, when Scott wants to open his text file ‘Report.txt’, he will be asked to input the user password for this file first. The system checks if it is the same as the one Scott gave at the encryption time, which is ‘TigerTxt’ and previously stored in SGDBM database. If not, the system will stop the execution. If it is correct, it will decrypt the file password and then use it to decrypt the file in the buffer in the system memory. Then it will check if the file has been tampered by SHA-1. If not, the decrypted file in the system memory will overwrite the file ‘Report.txt’ on disk. Otherwise, the system will stop the execution and keep the encrypted file on disk intact.

4.5.1 Implementation of some Common Functions

In CSFS, six commands that can be used as any other system commands, as well as a simple GUI application, were developed. They were designed for secure file operations and they are sfs_save, sfs_open, sfs_delete, sfs_rename, sfs_savedir, sfs_opendir and sfs_gui. Before presenting the implementation of the commands and the GUI application, it is necessary to present some functions invoked by the commands in common. These functions are listed below.

Function:

*char * GetHarddiskKey()*

*char * GetPSNKey()*

*char * GetBlowfishKeyForFileKey()*

*char * GetFileKeyForEncryptFile()*

As mentioned previously, the key used to encrypt a file password is made up of three parts: the serial number of the first IDE hard disk, the processor serial number (PSN) of the CPU currently used and the user password inputted directly by the user. The first two functions are used to get the first two parts of the key. In the third function, it first asks the user to input the user password and then, invoking the first two functions, generate the entire key that is used to encrypt a file password. The last function is used to ask the user to input a file password directly from the keyboard at the encryption time.

Function:

*gdbm_file_info * OpenDBFile(char OpenMode);*

*CloseDBFile(gdbm_file_info * gdbm_file);*

*int StoreIntoDB(char * FileName, int FileNameSize, char * FileKey, int FileKeySize);*

*int FetchFromDB(char * FileName, int FileNameSize, char * FileKey);*

*int DeleteFromDB(char * FileName, int FileNameSize);*

*int RenameKeyInDB(char * OldFileName, int OldFileNameSize, char * NewFileName, int NewFileNameSize);*

These functions involve in database operations. The first two functions are used to open or close a SGDBM database. The last four functions invoke the corresponding external functions provided by SGDBM to process key / data pairs for each file in CSFS. These four functions are used by the four *sfs* commands respectively.

Function:

*char * GetDBDir();*

*char * GetAbsFileName(char * RelFileName);*

In CSFS, each user has a unique SGDBM database to store file passwords. The data file must be put in the user's home directory and is created automatically at the first time saving a key / data pair in the database. The first function is used to obtain the user's home directory so that the database can be created in the correct directory. The second function is used to get the absolute path of a file. When users run the commands in CSFS, they do not need to give absolute directory for a file themselves. This function can check the file's relative directory that users provide and figure out the file's absolute directory. The file name, which is stored into the SGDBM database, must be the file's absolute path to keep consistence all the time.

4.5.2 Implementation of the Command: *sfs_save*

In CSFS, when a user needs to secure a file, he can run the command *sfs_save* followed by the file name. After the execution of this command, the file on disk will be encrypted and the file name will be changed to include a suffix '.sfs'. Users are always able to recognize the files that have been processed by CSFS by the suffix '.sfs'.

The implementation of this command can be divided into several steps as shown in the Figure 1. In the main function, it follows all steps in the figure by invoking related functions. In CSFS, each encrypted file has a corresponding key / data pair stored in the user's SGDBM database, which is the absolute file path / hash value plus the encrypted file password. The function used here is:

```
int StoreIntoDB(char * FileName, int FileNameSize, char * FileKey, int FileKeySize);
```

Once the corresponding key / data is stored into the SGDBM database successfully, the encrypted file is going to be saved back on disk and the file name is changed to include a suffix '.sfs'. This is implemented in the function:

```
int rename(char * OldFileName, char * NewFileName);
```

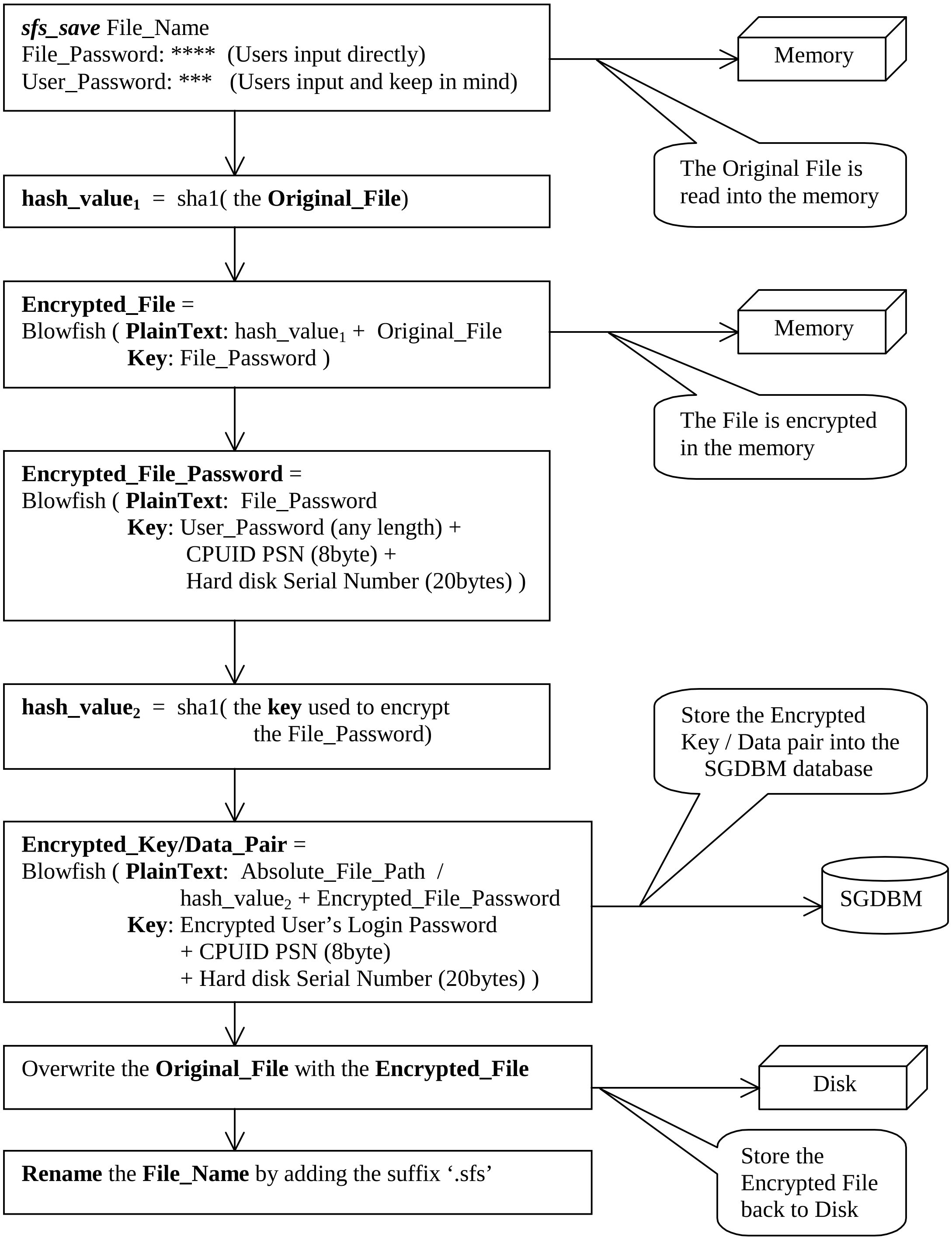

sfs_save File_Name
File_Password: **** (Users input directly)
User_Password: *** (Users input and keep in mind)
Memory
The Original File is read into the memory
hash_value$_1$ = sha1(the Original_File)
Encrypted_File =
Blowfish (PlainText: hash_value$_1$ + Original_File
Key: File_Password)
Memory
The File is encrypted in the memory
Encrypted_File_Password =
Blowfish (PlainText: File_Password
Key: User_Password (any length) +
CPUID PSN (8byte) +
Hard disk Serial Number (20bytes))
hash_value$_2$ = sha1(the key used to encrypt the File_Password)
Store the Encrypted Key / Data pair into the SGDBM database
Encrypted_Key/Data_Pair =
Blowfish (PlainText: Absolute_File_Path /
hash_value$_2$ + Encrypted_File_Password
Key: Encrypted User's Login Password
+ CPUID PSN (8byte)
+ Hard disk Serial Number (20bytes))
SGDBM
Overwrite the Original_File with the Encrypted_File
Disk
Store the Encrypted File back to Disk
Rename the File_Name by adding the suffix '.sfs'

Figure 1 command *sfs_save* flow chart

4.5.3 Implementation of the Command: *sfs_open*

When a user wants to access a file, in CSFS, he needs to open this file first. Users can run the command *sfs_open* followed by the file name. After the execution of this command, the whole file on disk is decrypted and the file name is changed to its original name without the suffix '.sfs'. Once a file is opened, it can be accessed as in a traditional file system, but it is not secure any longer. To make it secure, after accessing the file, users should run the command *sfs_save* again to make the file secure.

The implementation of this command can be divided into several steps as shown in the Figure 2. In the main function, it follows all steps in the figure by invoking relative functions. For each file, there is a corresponding key / data pair fetched from the user's SGDBM database, which is the absolute file path / hash value plus the encrypted file password. The function used here is:

```
int FetchFromDB(char * FileName, int FileNameSize, char * FileKey);
```

Once the key / data pair is fetched from the database successfully, the encrypted file will be decrypted in the system memory and the hash value of the original file is recalculated to check if it is same as the old hash value. If both hash values are same, the decrypted file is going to be saved back on disk and the file name will be changed back to its original name without the suffix '.sfs'. This is implemented in the function:

```
int rename(char * OldFileName, char * NewFileName);
```

In addition, the corresponding key / data pair will be deleted from the SGDBM database once the file is decrypted and saved back on disk. The function invoked here is:

```
int DeleteFromDB(char * FileName, int FileNameSize)
```

If the key / data pair is not successfully deleted, it will affect the user to save this file in the future. The file name has to be changed before the user saves this file again using the command *sfs_save.*

4.5.4 Implementation of the Command: *sfs_delete*

In CSFS, the command *sfs_delete* is used to delete files with the suffix '.sfs'. First, the system checks if the file name includes the suffix '.sfs'. If not, it will exit without deleting the file. If yes, the following function will be invoked to really delete the file from disk.

```
Int remove(char * FileName);
```

Once the file is deleted from disk, the key / data pair corresponding to this file will be deleted from the SGDBM database. The function invoked is:

```
int DeleteFromDB(char * FileName, int FileNameSize);
```

If the key / data pair is not successfully deleted, it will affect the user to save a new file that has the same name as the deleted file. The flow chart for the command *sfs_delete* is in the Figure 3.

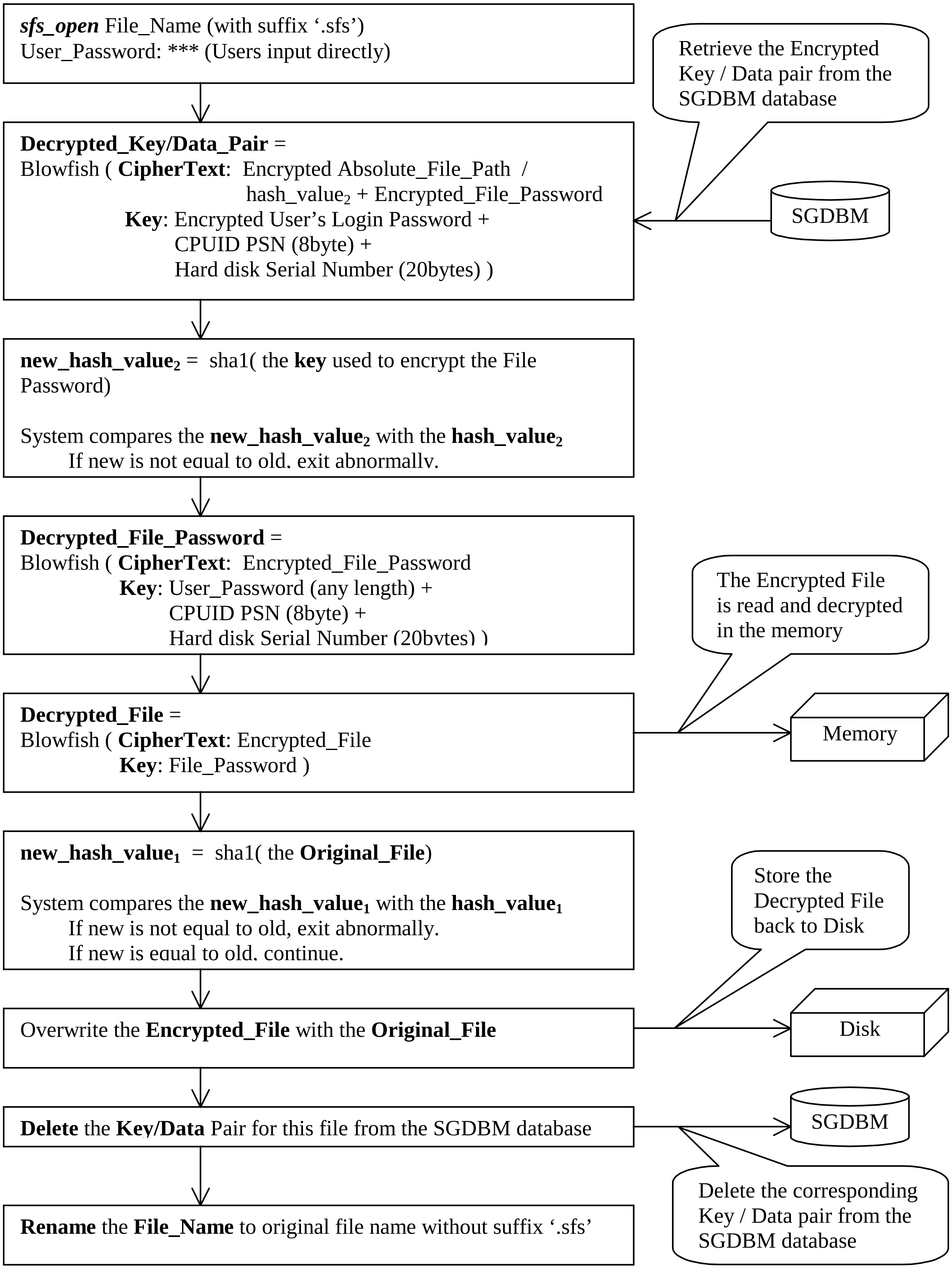

sfs_open File_Name (with suffix '.sfs')
User_Password: *** (Users input directly)
Retrieve the Encrypted Key / Data pair from the SGDBM database
SGDBM
Decrypted_Key/Data_Pair =
Blowfish (CipherText: Encrypted Absolute_File_Path / hash_value$_2$ + Encrypted_File_Password
Key: Encrypted User's Login Password + CPUID PSN (8byte) + Hard disk Serial Number (20bytes))
new_hash_value$_2$ = sha1(the key used to encrypt the File Password)
System compares the new_hash_value$_2$ with the hash_value$_2$
If new is not equal to old, exit abnormally.
Decrypted_File_Password =
Blowfish (CipherText: Encrypted_File_Password
Key: User_Password (any length) + CPUID PSN (8byte) + Hard disk Serial Number (20bytes))
The Encrypted File is read and decrypted in the memory
Decrypted_File =
Blowfish (CipherText: Encrypted_File
Key: File_Password)
Memory
new_hash_value$_1$ = sha1(the Original_File)
System compares the new_hash_value$_1$ with the hash_value$_1$
If new is not equal to old, exit abnormally.
If new is equal to old, continue.
Store the Decrypted File back to Disk
Overwrite the Encrypted_File with the Original_File
Disk
Delete the Key/Data Pair for this file from the SGDBM database
SGDBM
Delete the corresponding Key / Data pair from the SGDBM database
Rename the File_Name to original file name without suffix '.sfs'

Figure 2 command *sfs_open* flow chart

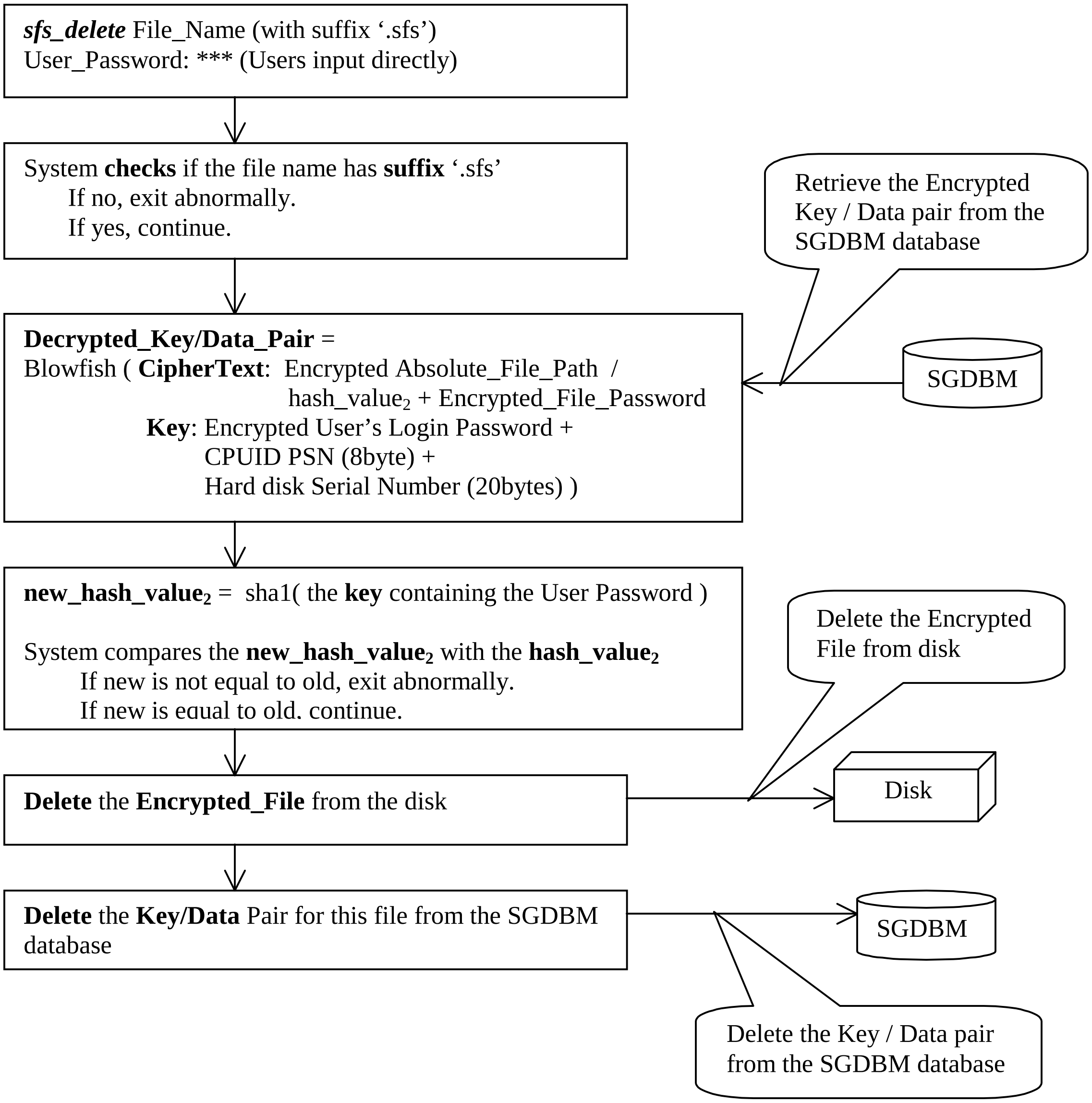

sfs_delete File_Name (with suffix ‘.sfs’)
User_Password: *** (Users input directly)
System **checks** if the file name has **suffix** ‘.sfs’
If no, exit abnormally.
If yes, continue.
Retrieve the Encrypted Key / Data pair from the SGDBM database
Decrypted_Key/Data_Pair =
Blowfish (**CipherText**: Encrypted Absolute_File_Path / $hash\_value_2$ + Encrypted_File_Password
Key: Encrypted User’s Login Password + CPUID PSN (8byte) + Hard disk Serial Number (20bytes))
SGDBM
$new\_hash\_value_2$ = sha1(the **key** containing the User Password)
System compares the **$new\_hash\_value_2$** with the **$hash\_value_2$**
If new is not equal to old, exit abnormally.
If new is equal to old, continue.
Delete the Encrypted File from disk
Delete the **Encrypted_File** from the disk
Disk
Delete the **Key/Data** Pair for this file from the SGDBM database
SGDBM
Delete the Key / Data pair from the SGDBM database

Figure 3 command *sfs_delete* flow chart

4.5.5 Implementation of the Command: *sfs_rename*

In CSFS, when a user wants to change a file's name or move a file to another directory, he can use the command *sfs_rename*. First, the system checks if the file name has the suffix '.sfs'. If not, it will exit without renaming the file name. If yes, the following function is invoked to rename the file.

*int rename(char * OldFileName, char * NewFileName);*

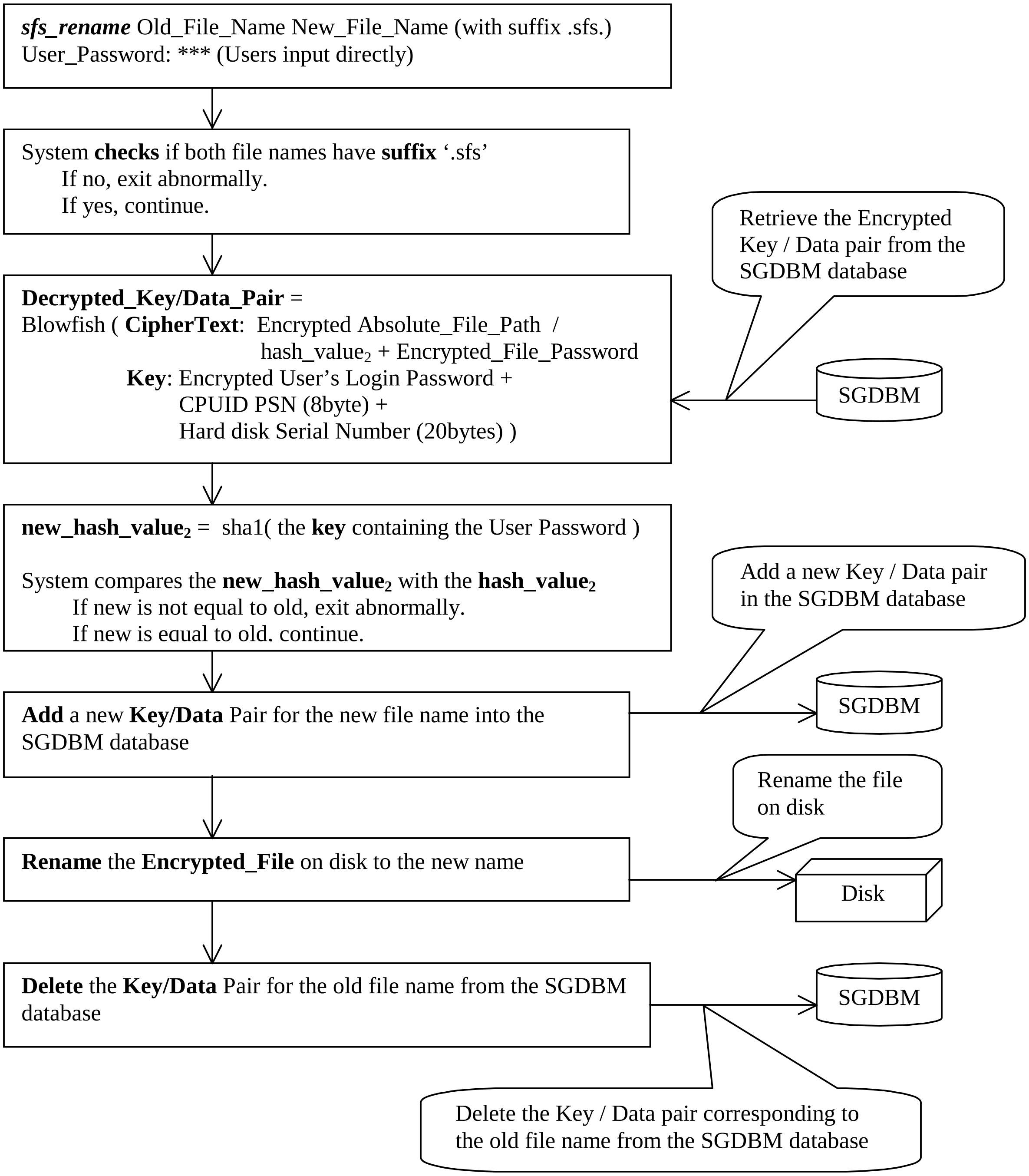
Before the file is renamed or moved, a new key / data pair corresponding to the new file name will be added into the SGDBM database. The key is the new absolute path of the file and the associated data keep intact. The function invoked is:

*int RenameKeyInDB(char * OldFileName, int OldFileNameSize, char * NewFileName, int NewFileNameSize);*

Once the file on disk is renamed or moved successfully, the key / data pair corresponding to the old file name will be deleted from the SGDBM database. The function invoked is:

*int DeleteFromDB(char * FileName, int FileNameSize)*

If the key / data pair corresponding to the old file name is not successfully deleted, it will affect the user to save a new file with the old file name in the future. The flow chart for the command *sfs_rename* is in the Figure 4.



sfs_rename Old_File_Name New_File_Name (with suffix .sfs.)
User_Password: *** (Users input directly)
System **checks** if both file names have **suffix** ‘.sfs’
If no, exit abnormally.
If yes, continue.
Retrieve the Encrypted Key / Data pair from the SGDBM database
Decrypted_Key/Data_Pair =
Blowfish (**CipherText**: Encrypted Absolute_File_Path / hash_value$_2$ + Encrypted_File_Password
Key: Encrypted User’s Login Password + CPUID PSN (8byte) + Hard disk Serial Number (20bytes))
SGDBM
new_hash_value$_2$ = sha1(the **key** containing the User Password)
System compares the **new_hash_value$_2$** with the **hash_value$_2$**
If new is not equal to old, exit abnormally.
If new is equal to old, continue.
Add a new Key / Data pair in the SGDBM database
Add a new **Key/Data** Pair for the new file name into the SGDBM database
SGDBM
Rename the file on disk
Rename the **Encrypted_File** on disk to the new name
Disk
Delete the **Key/Data** Pair for the old file name from the SGDBM database
SGDBM
Delete the Key / Data pair corresponding to the old file name from the SGDBM database

Figure 4 command *sfs_rename* flow chart

4.5.6 Implementation of the Commands: *sfs_savedir* and *sfs_opendir*

In CSFS, when a user wants to make a directory with files and subdirectories secure, he can use the command *sfs_savefile*. When the user need to access this directory, just running the command *sfs_openfile* can have all files and subdirectories opened at a time.

These two commands were implemented as shell scripts, which invokes some system commands to deal files and subdirectories and the corresponding *sfs* commands *sfs_save* and *sfs_open* to save and open files in the given directory. One thing necessary to be mentioned is that the file passwords for each file in the directory will be generated randomly but not inputted by the user. The detailed implementations are not presented here.

4.5.7 Implementation of the GUI Application: *sfs_gui*

Besides the six commands presented in the above sections, a simple Graphical User Interface (GUI) application was implemented for more friendly and easier use. All available GUIs, shown in the Figure 5, are used for all secure file operations in CSFS. The GUI application was developed with pure GTK+ in Linux.

GTK+ is a very capable cross-platform development technology and makes developing a GUI easy and fun [GTK00]. GTK+ is a library that can manage all aspects of developing a GUI application and which was initially designed to produce the GUI on just one – GIMP. It's a full object oriented library and provides a number of objects to manage any aspect of a graphical display. These include windows, widgets (the controls that are placed inside a window to draw the user interface) as well as signals – the events that occur in code when a user interacts with a widget in the application's user interface.

First, the corresponding package was installed into Linux so that the GTK+ libraries are available for use. Then, some code was written to draw all GUIs including some windows and necessary widgets. The code of the four *sfs* commands was modified to be the handler functions, which can be invoked when certain signals occur. Finally, some code was added to deal with all widgets, signals and handlers to make the whole application executable. After the compilation, users can execute the GUI application *sfs_gui* to do all secure file operations in CSFS.

5. User's Manual

In the following sections, how to install and use both the SGDBM and the CSFS will be presented in detail. Users should follow these steps to have the system run correctly and efficiently.

5.1 How to Install the System

To use CSFS, the whole system must be installed following the steps presented below. Only administrators can get the system installed correctly because there are some operations that need the administrator's privilege. Once the whole system is installed correctly, all users can use it in the same way. The installation consists of three major parts: recompiling the Linux kernel, installing the SGDBM and installing the CSFS.

5.1.1 Recompile the Linux Kernel

In both the SGDBM and the CSFS, when encrypting or decrypting any data using Blowfish, the processor serial number (PSN) of the CPU currently used is always used as a part of the key for encryption or decryption. As previously mentioned, in Linux, it is forbidden to obtain the PSN by default. Administrators have to do something special to enable the PSN on computers. Unfortunately, they have to have this feature enabled at boot time by modifying a specific kernel file a little bit.

Normally, the source code files of Linux are not installed on computers. Therefore, administrators must firstly check if the subdirectory including source files exists already.

Usually, it is under the directory */usr/src/linux*. If it does not exist, administrators have to add this package. Then, it is necessary to locate a specific kernel file *setup.c* in this directory. It is usually in the directory */usr/src/linux/arch/i386/kernel*.

Once the file *setup.c* is located, administrators need to open it in a text editor and modify a little bit code. To enable the PSN, a specific bit in a specific MSR needs to be set. By default, this bit is set to 1 at boot time to disable the PSN. The relative line in this file is:

```
lo |= 0x200000;
```

What users need to do is just modifying this line as below:

```
lo |= 0x000000;
```

Thus, the specific bit in the specific MSR is going to be set to 0 at boot time so that the PSN can be enabled and accessed on users' computers.

After modifying the kernel file *setup.c*, to make it take effect, recompiling the Linux kernel is inevitable. However, the steps may vary a little depending on the different Linux versions and variable system configurations. The overall steps are as follows:

```
cd  /usr/src/linux
make  config   ( or make menuconfig )
make  dep
make  zImage
( make  bzImage   -- for compressed kernel )
cp  /usr/src/linux/arch/i386/boot/zImage /boot/newkernel
```

The commands *make config* and *make menuconfig* are the classic commands to set up the system configuration before compiling the programs. In fact, administrators do not need to change any configuration. But, it is better to run one of these commands before recompiling the Linux kernel. The command *make dep* builds the tree of interdependencies in the kernel somewhat based on the configurations set in the first step. Both commands *make zImage* and *make bzImage* really recompile and build the actual kernel. The last step installs the new kernel. Administrators just need to copy the new kernel file into the directory including the boot file in Linux. Finally, rebooting the computer can make the new kernel functional. This way, the PSN will be enabled at boot time and will be able to be used as a part of the key for Blowfish.

In case that the Linux kernel cannot be recompiled successfully or the PSN cannot be enabled at any rate, in both the SGDBM and the CSFS, this part of the key for Blowfish will be replaced by a fixed data what is set in programs so that the whole system still can run normally. However, the security is going to be at a little lower level.

In addition, by default, users do not have the permission to open the hard disk file. Therefore, users are not able to access the serial number of the hard disk. Administrators need to run the following command to add the relevant permission to all users.

```
chmod +or /dev/hda
```

5.1.2 Install the SGDBM

After recompiling the Linux kernel, administrators need to install the SGDBM system. It is the base of CSFS. All source code files are put in one directory. Administrators should first copy this directory to one place on computers, then configure and compile all source code files in this directory. The steps to be followed are:

```
configure
make  install
make  distclean
```

The command *make install* generates some shared library files and then puts them into the system library directory what is usually */usr/local/lib*. Most of the functions provided by this shared library will be invoked by CSFS.

5.1.3 Install the CSFS

After installing the SGDBM system, administrators can start to install CSFS. Like the SGDBM, all source code files are put in just one directory. Administrators first need to copy this directory to one place on computers. Then one of the commands: *make*, *make cmd* or *make gui* should be run to compile the source code files and get the system installed. Running *make* can have both the six commands and the simple GUI application installed in the directory */bin* in Linux. They are:

```
sfs_save
sfs_open
sfs_delete
```

sfs_rename

sfs_savedir

sfsf_opendir

sfs_gui

Running *make cmd* will install only the six commands. Running *make gui* will install the GUI application *sfs_gui* only.

The six commands can be executed by just inputting the commands with all required parameters, like any other system command such as *ls*, *cd*, etc. Users can run these commands, at any time, in any directory without the necessity to give the absolute path of these commands.

5.2 How to Use the CSFS

After installing the CSFS successfully, all users can use the CSFS system at any time when needed. When using the six *sfs* commands, users must conform to their formats.

When a user wants to make a file secure, no matter whatever file type, just run the command below:

```
sfs_save  file-name
```

On executing this command, the user will be asked to input two passwords. The first one is the file password that is used to encrypt the file. The file password can be any length up to 56 bytes and needs not be remembered. The second password is the user password that is used to encrypt the file password. The user must keep the user password in mind because it is required when decrypting this file in the future. If some error raises, the execution will be stopped and the file will not be encrypted. Once the command is executed successfully, the file will be stored on disk in encrypted form and the file name will be changed to include a suffix '.sfs'.

When a user wants to open a file that has already been saved in CSFS, just run the command below:

```
sfs_open   file-name
```

On executing this command, the user will be asked to input the user password. The system will first check the file name given by the user. If the file name does not have the suffix '.sfs', the system will issue a warning message and exit normally. If the file name has the suffix, then the system will check the user password. If the user inputs a wrong user password, which is different from the one inputted at the encryption time earlier, the system will issue a warning message and exit normally without doing anything. If the user password is correct, the file will be decrypted to its original format and the file name will be changed to the original one without the suffix '.sfs'. However, if either the file password stored in the SGDBM database or the file have been tampered by somebody, the execution will stop and the encrypted file on disk will keep intact.

When a user wants to delete a file that has already been saved in CSFS, just run the command below:

sfs_delete file-name

On executing this command, the user will be asked to input the user password. The system will first check the file name given by the user. If the file name does not have the suffix '.sfs', the system will issue a warning message and exit normally. If the file name has the suffix, then the system will check the user password. If the user password is not correct, the system will issue a warning message and exit normally without doing anything. If the user password is correct, the file will be deleted from disk and then the key / data pair corresponding to this file will be removed from the SGDBM database.

When a user wants to, in CSFS, rename a file or move a file to another directory, just run the command below:

sfs_rename old-file-name new-file-name

On executing this command, the user will be asked to input the user password. The system will first check both the old file name and the new file name given by the user. If either of these file names does not have the suffix '.sfs', the system will issue a warning message to tell this fact and exit normally. If both file names have the suffix, then the system will check the user password. If the user password is not correct, the system will issue a warning message and exit normally without doing anything. If the user password is correct, the file name will be changed to the new one. Moreover, the key of the key / data pair corresponding to the file in the SGDBM database will be modified by the new file name.

When a user wants to make a directory secure, just run the command below:

sfs_savedir directory

On executing this command, the user will be asked to input just one password: user password. The file passwords for all files in the given directory will be generated randomly by system. The user must remember the user password because it is required when decrypting this directory in the future. Once the command is executed successfully, all files will be stored on disk in encrypted form and all file names will be changed to include the suffix '.sfs'.

When a user wants to open a directory that has already been saved in CSFS, just run the command below:

sfs_opendir directory

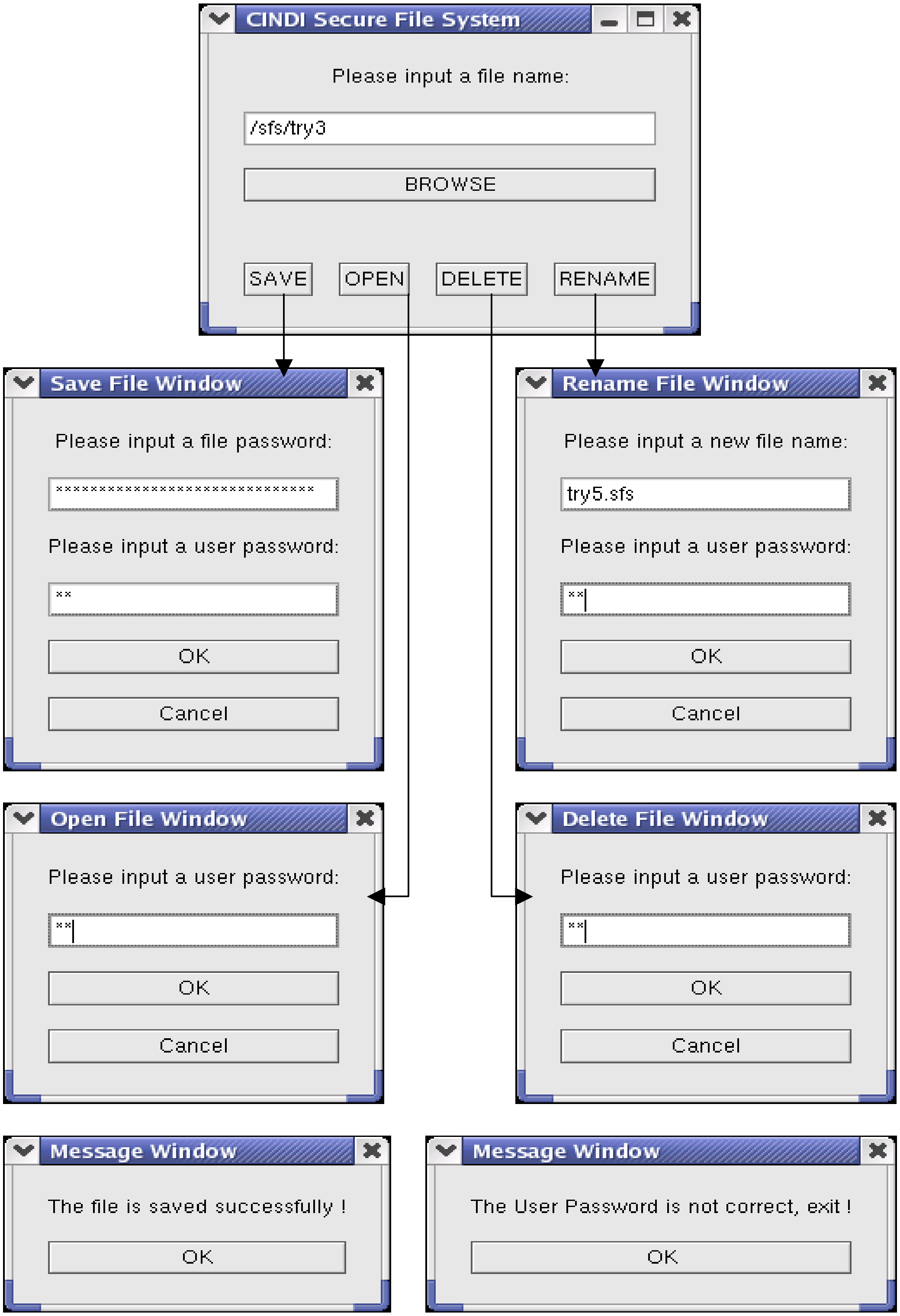
On executing this command, the user will be asked to input the user password. If the user inputs a wrong user password, which is different from the one inputted at the encryption time earlier, the system will issue a warning message and exit normally without doing anything. If the user password is correct, all files in the given directory will be decrypted to their original format and all file names will be changed to the original ones without the suffix '.sfs'.

In CSFS, besides the above six commands, users can also run a simple GUI application to do the secure file operations besides saving and opening directories. The command is:

sfs_gui

All GUIs may show up are in the Figure 5. Using this GUI application is very easy. First, the user should input a file name or select a file from the file dialog, which can be opened by pressing the button ‘Browse’. Then, according to the user’s need, press one of the four buttons: ‘SAVE’, ‘OPEN’, ‘DELETE’ and ‘RENAME’. A new dialog will pop up. The user needs to input the required password or file name and then press the button ‘OK’ to get the required file operation done. The message dialog will pop up and indicate the execution status.

One more thing has to be mentioned here is that each user has a unique SGDBM database in his home directory. The data file is always created at the first time the user saves a file in CSFS. The name of the data file is *junk.gdbm* and is same for each user. Once the database is created, the data file *junk.gdbm* will not be deleted automatically even though there is no file in CSFS. The user can delete it directly when it is no longer used. Or, whenever the administrator removes the user from the system, the data file will disappear together with the user’s home directory.



CINDI Secure File System

Please input a file name: /sfs/try3

Save File Window

Please input a file password: *****************************

Please input a user password: **

Rename File Window

Please input a new file name: try5.sfs

Please input a user password: **

Open File Window

Please input a user password: **

Delete File Window

Please input a user password: **

Message Window

The file is saved successfully !

Message Window

The User Password is not correct, exit !

Figure 5 application *sfs_gui* GUIs

5.3 How to Use the SGDBM

The SGDBM was implemented as a shared library as any other library in Linux. It provides some external functions that can be invoked by programmers. All of these functions were presented in the section 4.4.2 and are not presented here again. The CSFS invokes these external functions to perform necessary secure database operations. If need be, any other programmer can use this shared library to do their database work.

6. Contributions, Conclusions and Future Work

In the previous chapters, the implementation of both the SGDBM and the CSFS and how to use both systems as well as some relevant technologies were presented. This chapter will summarize these two systems and discuss some existing problems and corresponding future work that may improve both secure systems.

6.1 Contributions and Conclusions

In this report, the implementation of two secure systems was presented. One is the SGDBM, which is a simple Secure Database Management System. It deals with both data and metadata in a database uniformly in a low-level model. All content in a data file of SGDBM is protected by encryption and is secure all the time. Whenever anything in a database has been tampered or corrupted by some attackers, the SGDBM is able to find out the unexpected modifications and give certain warning messages. The other secure system implementation is the CSFS. It provides six commands, which can be used by users in the same way as any other basic command in Linux, and a simple GUI application. This Secure File System depends on SGDBM to perform all necessary secure database operations such as storing or fetching a file password associated with a file. The connection between these two secure systems is completely transparent to users.

One of the author's contributions, in this project, was to study the existing GDBM system in Linux and transform it to be SGDBM, a secure DBMS, by modifying related source codes. The other contribution was that the author designed and implemented the entire CSFS system.

6.2 Problems and Future work

Although both the SGDBM and the CSFS were implemented, they are elementary and not very complete. Some questions are still open. The sections below will discuss some parts of the implementation that may be improved in the future.

6.2.1 Future Work for Secure DBMS

The SGDBM was implemented based on GDBM in Linux. It is functionally restricted by the internal limitation of the GDBM system itself. Because the GDBM in Linux is an elementary database management system and does not provide all the functions that a DBMS is supposed to give. Firstly, it does not support backup and recovery. Secondly, it does not provide index management. Lastly, it does not support all kinds of objects that usually exist in a commercial DBMS. Each individual GDBM database is, in fact, a table consisting of only two columns, which is a key / data pair.

With the above limitations of GDBM, even though the goal to develop a Secure DBMS was accomplished, it is just an elementary secure database management system and can be used only by some not-too-complicated database applications. In the future work, this database system can be improved by adding more database functions. Or, a new Secure DBMS can be created by applying the approach of establishing a secure database system at low level.

6.2.2 Future Work for Secure File System

CSFS provides users some commands for most frequently used file operations. Users can use them in the same way as using any other basic command in Linux in any directory without the necessity to give the command’s directory. In CSFS, a user does not need to encrypt all his files. The User can execute the command *sfs_save* to encrypt the files, which need to be secured. Whenever a file is not necessary to be secure, the user can execute the command *sfs_open* to restore the file into its original un-encrypted status. These properties of CSFS give users more flexibility to deal with their files. However, the disadvantage is that it is not transparent to users as some other Secure File System presented in the first chapter.

In addition, the CSFS was implemented on the per-file basis. This is another type of limitation as well. When users want to access only a small part of a large file, the system will have the whole file processed. This may bring more overheads due to the requirement for more disk I/O operations. Another disadvantage is that the CSFS is not a complete file system because it does not provide some functions that a file system is supposed to give, such as processing file permissions.

In the future, all the disadvantages mentioned above are possible to be overcome. Applying the idea of CSFS, it can be attempted to directly modify the Linux kernel to make the file system of Linux to be a secure system. Thus, all of the files in Linux will be secured at any time. However, this will bring more overheads by encryption and some other operations.

REFERENCES

[BIOS00] "BIOS Chip Basis" http://diybios.51.net/biosic/right3.htm

Latest accessed on Aug.15, 2003

[CPUID1] "The CPUID Guide" http://www.paradicesoftware.com/specs/cpuid/index.htm

Latest accessed on Aug.15, 2003

[CPUID2] "IA-32 architecture CPUID" http://www.sandpile.org/ia32/cpuid.htm

Latest accessed on Aug.15, 2003

[EAC01] "Encryption Algorithms"

http://www.mycrypto.net/encryption/crypto_algorithms.html

Latest accessed on Aug.15, 2003

[EAC02] "General Cryptography-related Frequently Asked Questions"

http://www.f-secure.com/support/technical/general/crypto.shtml

Latest accessed on Aug.15, 2003

[EFS00] "Encrypting File System: Your Secrets are Save"

http://www.microsoft.com/windows2000/techinfo/planning/security/efssteps.asp

Latest accessed on Aug.15, 2003

[GART97] "Architecture of the Secure File System"

http://storageconference.org/2001/2001CD/p01hughe.pdf

Latest accessed on Aug.15, 2003

[GDBM03] "Gdbm" http://www.delorie.com/gnu/docs/gdbm/gdbm_toc.html

Latest accessed on Aug.15, 2003

[GPL001] "The GNU General Public License (GPL) Version 2, June 1991"

http://www.opensource.org/licenses/gpl-license.php

Latest accessed on Aug.15, 2003

[GTK00] Peter Wright "Beginning GTK+/GNOME Programming"

ISBN 1-861003-81-1

[HWEE03] "StegFS: A steganographic File System"

Hwee Hwa Pang, Kian Lee Tan, Xuan Zhu

Proceedings of the 19th International Conference on Data Engineering

Bangalore India 2003 PP.657-668

[MLS00] "Multi-level Secure Database Management Schemes"

http://www.sei.cmu.edu/str/descriptions/mlsdms_body.html

Latest accessed on Aug.15, 2003

[MICH00] "Secure File System SFS" http://atrey.karlin.mff.cuni.cz/~rebel/sfs/

Latest accessed on Aug.15, 2003

[PSN01] "PSN" http://www.sims.berkeley.edu/courses/is224/s99/GroupG/psn_wp.html

Latest accessed on Aug.15, 2003

[SFS00] "Secure File System" http://www.securefs.org

Latest accessed on Aug.15, 2003

[SHA195] "Secure Hash Standard" http://www.itl.nist.gov/fipspubs/fip180-1.htm

Latest accessed on Aug.15, 2003

[SCHN93] Bruce Schneier

"Applied Cryptography Second Edition: protocols, algorithms, and source code in C"

ISBN 0-8493-8523-7 Publisher: New York: Wiley, c1996

[UMES02] "How to Build a Trusted Database System on Untrusted Storage"

http://www.usenix.org/events/osdi2000/maheshwari.html

Latest accessed on Aug.15, 2003

All above web references are archived at:

http://www.cs.concordia.ca/~bcdesai/grads/ying_liu/reference

Also the software is accessible from:

http://www.cs.concordia.ca/~bcdesai/grads/ying_liu/csfs