Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

ImgFS: a transparent cryptography for stored images using a filesystem in userspace*

Osama A. KHASHAN‡, Abdullah M. ZIN, Elankovan A. SUNDARARAJAN
(*Centre for Software Technology and Management, Faculty of Information Science and Technology, National University of Malaysia (UKM), Bangi 43600, Selangor, Malaysia*)
E-mail: o_khashan@yahoo.com; amz@ftsm.ukm.my; elan@ftsm.ukm.my
Received Apr. 8, 2014; Revision accepted Oct. 9, 2014; Crosschecked Dec. 11, 2014

Abstract: Real-time encryption and decryption of digital images stored on end-user devices is a challenging task due to the inherent features of the images. Traditional software encryption applications generally suffered from the expense of user convenience, performance efficiency, and the level of security provided. To overcome these limitations, the concept of transparent encryption has been proposed. This type of encryption mechanism can be implemented most efficiently with kernel file systems. However, this approach has some disadvantages since developing a new file system and attaching it in the kernel level requires a deep understanding of the kernel internal data structure. A filesystem in userspace (FUSE) can be used to bridge the gap. Nevertheless, current implementations of cryptographic FUSE-based file systems suffered from several weaknesses that make them less than ideal for deployment. This paper describes the design and implementation of ImgFS, a fully transparent cryptographic file system that resides on user space. ImgFS can provide a sophisticated way to access, manage, and monitor all encryption and key management operations for image files stored on the local disk without any interaction from the user. The development of ImgFS has managed to solve weaknesses that have been identified on cryptographic FUSE-based implementations. Experiments were carried out to measure the performance of ImgFS over image files' read and write against the cryptographic service, and the results indicated that while ImgFS has managed to provide higher level of security and transparency, its performance was competitive with other established cryptographic FUSE-based schemes of high performance.

Key words: Storage image security, Cryptographic file system, Filesystem in userspace (FUSE), Transparent encryption
doi:10.1631/FITEE.1400133 **Document code:** A **CLCnumber:** TP309.7

1 Introduction

Recently, digital images have played a more significant role in society, where technological progress in the last two decades has introduced a huge flow of digital images in storage devices. This, in turn, has introduced many digital images and image applications that in many circumstances provide vital information, such as military image databases, geography image sensing, medical imaging systems, scientific images, and personal image albums. However, data security in storage systems is a difficult problem since the lack of security provisions for disk-resident images can invite attackers to gain access and risk the privacy of sensitive data by compromising the security applied.

Cryptography is, so far, the most cost-effective solution to frustrate malicious attacks while preserving the security and confidentiality of these stored images. There are a large number of end-user encryption applications available to provide encryption utility for various users' file types. Unfortunately, most of such software encryption applications have been developed with a poor understanding of security concerns, or used standard security requirements that contain many states with legacy design choices that

‡ Corresponding author
* Project partly supported by the Ministry of Higher Education of Malaysia under Grant LRGS/TD/2011/UKM/ICT/02
ORCID: Osama A. KHASHAN, http://orcid.org/0000-0003-1965-1869
© Zhejiang University and Springer-Verlag Berlin Heidelberg 2015

are incompatible with the goal of data confidentiality (Mellado *et al.*, 2010).

Conventional performance is another key problem with such encryption applications. Performance failures occur when the encryption application is unable to meet the real-time requirement. The manual nature of encryption applications usually requires much user participation and may not be easy to use. The increased overhead incurred by the user provides unwanted liability and is an inconvenient mechanism that may introduce a dangerous encryption scheme, where routine use could make users careless or inadvertently leave files in clear view. Therefore, these storage encryption applications are recognized to be poor quality software protection and are always influenced by the security requirements.

Accordingly, many researchers place a focus on a transparent encryption since the minimal amount of user interaction to set up and use the system effectively increases the security and usability. This type of encryption mechanism can be implemented most efficiently with the operating system's file systems. It can offer a better functionality and avoid many of the pitfalls of using the user-level application programs, since the privileges given to all user space applications to protect the data would run with fewer privileges than those provided by the kernel level (Jaeger *et al.*, 2011).

Consequently, when the encryption operations are carried out as a basic part of the file systems, it can provide a flexible and transparent cryptographic solution. Furthermore, this approach can be used to incorporate advanced key management, authentication, and access control mechanisms.

Transparent encryption can be placed at different layers inside the kernel. It can be performed as a middleware level encryption by inserting a cryptographic layer between the user space and the real file system. Several middleware cryptographic file system schemes are available to provide real-time encryption and decryption services for different file types. Such schemes are based on either using the file system filter driver technology, which is an optional driver attached on a level between the I/O manager and the real file system driver in the windows kernel (Zhang *et al.*, 2009; Li and Jia, 2010; Khashan and Zin, 2013), or using a stackable file system in the Unix-like kernel (Cattaneo *et al.*, 2001; Wright *et al.*, 2003; Halcrow, 2005). This operates by encapsulating an encrypting file system and loading it at the top of the real file system to mount over one or more user's directories.

Transparent cryptography can also be performed at a block device level. This operates by inserting a cryptographic layer between the real file system and the device driver. The encryption driver works with blocks of raw data to encrypt the whole single or multiple disk partitions, like Cryptographic Disk (CGD) (Dowdeswell and Ioannidis, 2003), FreeBSD's GEOM (Schiesser, 2005), and work done by Singh *et al.* (2006) and Ma *et al.* (2010).

Although these kernel level cryptographic approaches can provide a robust solution for data security and a high level of transparency, they have their own inherent disadvantages since developing a new file system and attaching it in the kernel middleware level or the block device level is a challenging task. It depends on the specifics of the operating system and interacting components that require a deep understanding of the kernel internal data structure (Mazières, 2001). Furthermore, they do not support the system's portability or a file sharing application, and do not provide options for non-privileged users to mount and use the encrypting file system. On the other hand, a block device layer suffers from the fact that all data residing in the entire volume or disk need to be encrypted using a single key without any options for backup or access control for fine grain objects like individual files or directories. It also requires a pre-allocation of a fixed region on the disk (Ludwig and Kalfa, 2001).

Recently, operating system functions can provide users with a useful high-level application programming interface (API) to extend the capabilities of the file system in user space by expanding the operating system functionalities to develop their own file system without the need to change the underlying file system or to come out of the previous kernel level challenges and efforts (Rajgarhia and Gehani, 2010). A filesystem in userspace (FUSE) can bridge the gap in features between the application programs and the kernel file systems by extending the kernel level functionalities to the user space in order to provide services that are supported by a wide variety of applications, as well as enabling them to be performed in a transparent way.

In this paper, a new cryptographic file system called ImgFS has been developed, which is a user-level file system based on the FUSE technology (Szeredi, 2010) and the OpenSSL library (OpenSSL Project, 2014) for Linux, aiming to efficiently provide a cryptographic service for stored image files at a higher level of security, transparency, and flexibility. The development of ImgFS has also managed to overcome the weaknesses related to other cryptographic FUSE-based implementations.

2 Related work

Several cryptographic user space file system schemes have been developed from the earlier work on the cryptographic file system (CFS) (Blaze, 1993) to realize the cryptographic operations in a transparent way, such as CryptoFS (Hohmann, 2006), EncFS (Gough, 2008), and Chaoticfs (Shukela, 2013).

CFS was implemented in user space as a network file system (NFS) server. The user is required to create an encrypted directory on the local or remote file system to store his/her sensitive files inside, and after the user has been assigned the required key, the files are transparently encrypted. The CFS daemon, which acts as a pseudo NFS server, requires the authenticated user to use a special command to attach the encrypted source directory into a special directory called the mount point. This mount point acts as an un-encrypted window for the user to view his/her files in a clear form. At the end of the user session, by using a detach command, the clear text files automatically disappear. CFS uses a data encryption standard (DES) encryption algorithm in hybrid mode of electronic codebook (ECB) and output feedback (OFB) to encrypt files.

CryptoFS takes advantage of the Linux userland file system (LUFS) and FUSE, where CryptoFS can be built using both methods for the same encrypted directory. When CryptoFS is built for FUSE, it can let the program mount the file system. On the other hand, if it is built for LUFS, a shared library would be built to be used by the LUFS's daemon (lufsd) to mount the file system. In CryptoFS, the cryptographic library is based on the code from GNU privacy guard (GnuPG). It can support various encryption algorithms such as AES, DES, Blowfish, Twofish, Arcfour, and CAST5 using the cipher feedback (CFB) mode.

EncFS is the most well-known cryptographic user space file system with high performance. It was implemented using a FUSE library to provide a file system interface on the user space. When EncFS runs the file system over a specific directory, the user space daemon is improved to provide a transparent encryption and decryption for all files that are stored on the mounted directory. EncFS generates a per-block initialization vector (IV) by XORing the file IV with the block number, and a checksum MAC-header for each encrypted block. It uses the standard OpenSSL cryptographic library and supports AES and Blowfish block ciphers with several different options.

Chaoticfs is a FUSE-based encrypting file system that was implemented to encrypt user's files and append them in the storage as random blocks of data without timestamps or sequence numbers. Moreover, Chaoticfs can allow the user to partially expose his/her encrypted data using multiple entry points without statistical issues.

Unlike kernel-level encryption schemes, the existing cryptographic user space file system schemes provide transparent encryption for file content by permitting the non-privileged user to mount the encrypted file system without any required modifications into the internal kernel structure. They can be portable file systems as well as support remote access.

However, each of them has its own weakness that makes it less than ideal for them to be deployed. The first constraint of these schemes is the limited level of transparency provided for the user, where the user has to manually mount the file system and has to manually enter the passphrase needed for the source directory each time the directory is attached. Secondly, authenticating the legal user to mount the file system is based only on the supplied password and this is not sufficiently secure. Thirdly, all encryptions are done using a single key that is stored along with the data on the local disk. Fourthly, the user is required to remember all different passphrases associated with the various secret directories for a long time. Fifthly, the encryption is performed at a fine grain level of the directory, and once the file system is mounted over that directory, all of its files are decrypted with an extra performance penalty. Last but not the least, the

current schemes do not support multi-user systems as well as file sharing of protected resources.

3 System design

ImgFS is designed to mitigate or completely eliminate many issues that have not been addressed by the kernel level or FUSE-based cryptographic file system schemes. It is designed on the principle that the trusted system components and image files should be protected in a full transparent manner without a single user interaction. ImgFS's perspective is restricted only to the stored images to be forcibly encrypted or decrypted on-the-fly, where the user will not perceive the fact that he/she is using a different file system. Unlike previous cryptographic user space file system schemes, ImgFS can be placed to provide a sophisticated way to access, manage, and monitor all cryptographic and key management operations related to stored images with higher levels of security, authenticity, transparency, user convenience, and performance efficiency.

3.1 FUSE structure

The design structure of FUSE consists of various parts in both the user space and Linux kernel interfaces. Fig. 1 illustrates the interaction between these parts and ImgFS.

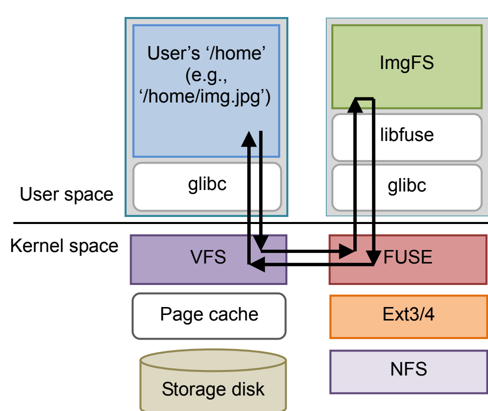

Fig. 1 Interaction between FUSE and ImgFS

When a user mounts the ImgFS encrypting file system to read or write an image file, the standard C library (glibc) dynamically generates a standard system call and passes it directly to the virtual file system (VFS) layer in the kernel. The VFS layer sits above the real file system to provide a uniform interface over different kernel file systems and storage devices. It abstracts the functionality of a file system, consults the mount table, and deals with all system calls targeted on the mounted file system according to its file path by checking out the pointer on its super block. If the system call is targeting the mounted ImgFS, VFS would subsequently turn it directly to the FUSE kernel module.

The FUSE kernel module, through the FUSE library (libfuse) on the user space, can allow the user to create his/her own file system implementation to add new features or enhance the current functionalities. It provides a developer with a list of standard system calls to handle the interaction communication between the VFS layer and the FUSE module to intercept the developed file system-related system calls, which can be parameterized with different security methods to allow a non-privileged user to mount and use them. Here, when a system call achieves the condition that is related to an image file and the image is a part of the mounted directory, the FUSE module receives the system call request and queues it into a local buffer until it is realized by the ImgFS daemon on the user space.

The libfuse consists of the main functions called by the ImgFS code. It is responsible for mounting ImgFS, initializing the data structure, setting up the mount point, and handling the communication between ImgFS, libfuse, and the FUSE module. Therefore, when the libfuse has realized the system calls in the FUSE local buffer, it subsequently invokes the callback requests from the FUSE module and then parses them to call up the relevant callback functions that are realized on ImgFS.

ImgFS has been implemented as a backend layer located in the user space. It interacts with standard file system calls, like open, read, write, and save. Therefore, by realizing the logical relation structure of the image file, it initializes the encryption algorithms provided by the OpenSSL library, and the encryption keys to provide a transparent encryption or decryption for the image file before the read or write operation takes place. Finally, when the cryptographic process has been completed, the result is sent back to the libfuse to return the response through the FUSE kernel module, either to the local disk for storage, or to

the user or the application issuing the system call in response to the image of the respective write and read requests.

3.2 Auto-mounting ImgFS

ImgFS is implemented as a user space resident file system to work with single- and multi-user based systems. The system administrator is responsible for the proper installation of ImgFS and for parameterizing the authentication polices of the non-privileged users during the mounted session. Each user has a personal login identification signature token, which is used for user authentication before mounting ImgFS and entering a secure session. The system checks the authenticity of the signature token issuer by dynamically loading it into a configured pluggable authentication module (PAM) (Sunsoft, 2014).

PAM is a Linux loadable module that provides a library of functions that are configured according to specific requirements determined by the system administrator to authenticate legal users and verify their permissions. However, PAM is more flexible in that it allows individual users or a group to log in a secure account or a secure session with different permissions and authentication criteria. Once the legal user has been authenticated, the system would automatically mount ImgFS over the entire user's home directory during the Linux login-session time. So, for any request to write or read an image file under the mounted home hierarchy tree, ImgFS can easily recognize and consider it for transparent encryption or decryption, while other non-image files under the home hierarchy will be left in plain view.

Internally, the kernel keeps information about the mounted ImgFS inside a mount table. Thus, the kernel would translate the file's path name and then consult the mount table. Therefore, through the ImgFS mounting time, every image path name would refer to the mounted ImgFS, whereas the original home directory which is under the root file system and stores the encrypted versions of the image files would no longer be accessible until ImgFS is unmounted. In such a design, the user is free to select any place under the '/home/' hierarchy tree to store his/her image files, which would be transparently encrypted to the opposite place on the original source home directory, which is distinguished with a suffix extension '/home.imgfs'.

The logic for this design decision is firstly to ensure that only a trusted user is able to mount ImgFS to enter a secure session without any required root privileges and without the need to issue attached commands or type in the same passphrase several times. Moreover, a user with root privileges is not trusted to enter a secure session with the encryption keys. Secondly, this design has removed the various versions of the mounted directories under the home hierarchy tree, since the entire user's home directory would become a mount point that is transparently mounted. This thus gives the user wider choices to store his/her image files and will not be restricted to a single directory place.

3.3 Authentication operation

Accessing a secure ImgFS session is bound to the Linux login session. Our system further duplicates the user authentication method through the PAM modular framework so as to have a stricter control over access to a secure session before the user can mount ImgFS. It is necessary to have a strong user authentication; if the attacker successfully guesses the user's account passphrase of the system login, the attacker would face another login authentication phase to gain entry into a secure ImgFS session. Of course, the user also must be sure that the file system would never reveal any protected data without authorization.

The authentication scheme in our system has been designed by deriving the user's signature token from his/her Linux account password. As such, the user's token must remain completely independent of the Linux password because of the possibility of malicious administrators and crackers (Lee and Ewe, 2007). The risk becomes great if the passphrase is directly chosen from the user, which is vulnerable to guessing attacks. Also, human memorability constrains the number of keys that a user can use.

Therefore, to provide enough entropy of the generated user's token and reduce the risk of compromise through user carelessness, the scheme transparently produces a salted value from a repeated character 's', which is then combined with the user's passphrase to generate a salted passphrase with at least 16 characters. Subsequently, the generated salted passphrase is hashed using MD5-128 (Rivest, 1992) to generate the user's identifying signature token.

When a user is to be defined, the user's signature token would be defined on the configured PAM module. In addition, the signature token is assumed for safekeeping on a smart card or USB device that belonged to the same user. A trusted platform module (TPM) (Trusted Computing Group, 2011) can also be a possible storage mechanism to ensure the security of the user's signature token in allowing both local and remote access to a secure session. Upon logging into the Linux account, the user's signature token must be retrieved dynamically from his/her removable media and directly assigned to the PAM module for authentication purposes. If the match with PAM's token version is realized, ImgFS will automatically mount. When a mounting connection is no longer needed, the entire '/home' mount point directory contents automatically disappear once ImgFS is un-mounted by the user or detached when a system is switched off. Fig. 2 demonstrates how the system maps the user's login passphrase to generate the signature token and authenticates him/her to mount ImgFS for a secure session entry.

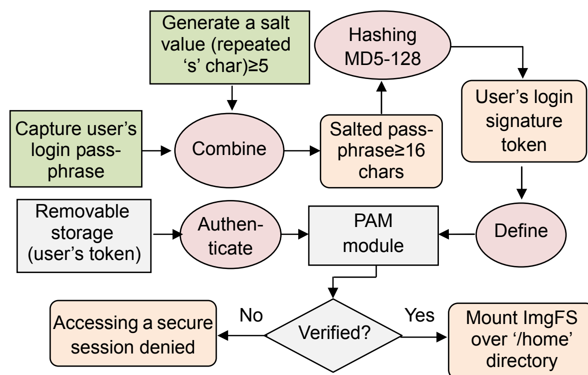

Fig. 2 User authentication processes to mount ImgFS

3.4 Key management operation

Users would never want to manipulate with a manual key management function and they usually prefer the more relaxed schemes that provide a transparent access solution. On the other hand, the strength of the encryption scheme is believed to be largely dependent on the strength of the policy used to protect the keys. A wrong key management policy might harm the system security or severely inconvenience the user. Thus, in ImgFS, this issue has been addressed by designing a more sophisticated key management scheme that operates in such a manner to be as transparent as possible to the user.

As part of ImgFS, the key management policy firstly results from the name of the image file that a user decides to access. Each image file has a unique file encryption key (FEK) that is used to encrypt the image file contents. Thereby, it is impossible to find two similar ciphered images corresponding to the same plain image. Obviously, if the attacker has successfully obtained FEK for one image file, he/she would not be able to recover the FEKs for other protected images.

The other key management policy is concerned with the user's identity. The key management operation binds a public encryption key (PEK) with the respective user's identity. Each user has his/her own PEK, which is used to encrypt his/her corresponding image FEKs. Therefore, the public key cryptography can ensure that image files are accessible only by an authorized user approved by ImgFS. However, the user cannot read or modify the image file without possessing an appropriate private encryption key (PrEK). Also, it makes the multi-user image file sharing operation easier.

Consequently, when the image file is newly created, a random 128-bit FEK would be generated to encrypt the image file contents. Here the random number generator supported by the Linux kernel is used (Kerrisk, 2013). Hence, the owner's PEK or any other user's PEK that the owner desires to share the image file with would be used to encrypt the FEK of the shared image file and subsequently append the encrypted FEK on the header of the image file.

To facilitate the image file sharing service, the share option is given and controlled by the right-click of a mouse button, which also displays the name list of registered system's users. Thereafter, the user's PEK would be hashed using MD5-128 and then stored hidden on the local disk, whereas a removable storage media, e.g., USB device or smart card, needs to be assumed as trusted storage for the user's PrEK to ensure that it is owned by the legitimate owner.

A file access verifier of a user's hashed PEK (HPEK) is used to verify the authenticity of the identity of the user who is trying to access the clear image data before opening or modifying the encrypted image, as well as to ensure that the image file has not been

tampered with by anyone previously. The generated access verifier is then stored along with the header of the image file.

3.5 Cryptographic operations

3.5.1 Image file formats

The development of a new digital image cryptosystem requires several issues to be taken into account. One of these is related to the intrinsic properties of a digital image of complex structure and huge size that is always greater than ordinary text. As such, the processing power with the computational complexity is always an overhead and this badly affects the efficiency and response time of the system. A further point is related to the various formats that are used for storing image files, where some of these image formats allow data compression to be applied on the image data (Khashan *et al.*, 2014).

Each file has a magic value used as a signature to recognize the file type, which is contained within other metadata in a file header (Kessler, 2014). ImgFS is based on these magic signatures with the ability to support a wide range of well-known image formats, both compressed and uncompressed.

3.5.2 Initialization vector

It was decided to break the image file into blocks of data consisting of 4096 bytes each, similar to the natural block size used in the Linux operating system. Each image file has a unique IV associated with it. When an image file is to be written on the local disk for the first time, a random file salt (FSalt) of 64-bit size would be generated to be used later to create the IV. To provide a strong protection for this FSalt, it should be encrypted with the user's PEK and then attached to the header of the image file.

The OFB mode was selected, as it can provide high-performance and high-security levels, and also the ability to hide all features of an image without increment on the size of the encrypted image (Preneel, 2011). Since the IV for the first block (IV0) in the OFB mode is used as a seed for all subsequent data blocks, this research decided to increase its level of difficulty. Consequently, the IV0 for each image file is created by a hash-based message authentication code (HMAC) of the image FEK and FSalt using HMAC-MD5 (Bellare *et al.*, 1996).

3.5.3 Encryption interaction

After a user meets the authentication criteria to access an active ImgFS session based on his/her granted permissions, the user can select any place under the home hierarchy tree to write his/her image file. Hence, ImgFS immediately responds to the user's callback request by creating a file access verifier of hashing the user's PEK and then appending it with the image file's header. Next, it would generate a random FEK and FSalt. The OpenSSL library was integrated with the ImgFS daemon to offer the cryptographic libraries, and the Blowfish encryption cipher was picked as a symmetric encryption algorithm of 128-bit key length. Blowfish has an efficient performance and was considered to be one of the fastest symmetric encryption algorithms (Verma *et al.*, 2011).

Following that, the image file is divided into a number of blocks of arbitrary length of 4096 bytes each. Hence, once IV is generated, the image blocks are directly encrypted on-the-fly. After the creation of the encrypted image blocks, the user's PEK would be used to encrypt FEK and FSalt by applying the RSA-1024 encryption algorithm, and appending them at the beginning of the encrypted image file's header.

On the other hand, if the encrypted image file is to be read from the disk, ImgFS would first verify the authenticity of the key holder's identity before reading the encrypted image. The verification is performed by loading the stored HPEK from disk and then extracting the stored file verifier on the header file, following which the comparison is performed. Once the keys are matched, the user's PrEK would be automatically loaded to decrypt FEK and FSalt. Thereupon, FEK and FSalt are made available to decrypt the entire image blocks. Fig. 3 shows the sequence of actions that take place when the encrypted image file is accessed.

We added an optional choice during the installation of ImgFS to protect the image file names, since file names can offer important information on the file's nature to an attacker, even if he/she cannot access the file data. We encode the image file names in an ASCII representation using a base64 encoder. Thus, we can ensure that the name remained intact without generating non-printable characters (which may be generated using other standard cipher algorithms),

and hence representing legal characters results in secure protection of the image names during storage on the underlying disk.

Since the encryption on ImgFS is associated on a per-image file basis, to allow the user with the ability to select the appropriate image file to be opened, we designed ImgFS so that during the mounting time, a transparent decoding for all image file names would be carried out to transform them back into their own clear text names. Thereby, if any update occurred on any image file's name during the mounting time, the update would be directly realized on its encoded counterpart on the source home directory.

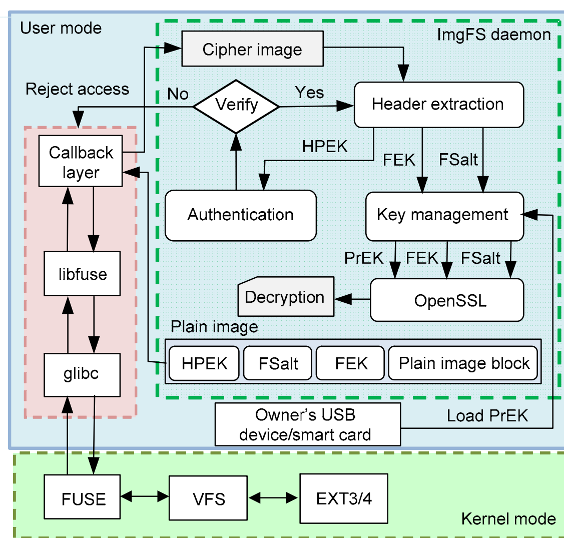

Fig. 3 Image file's decryption process in the ImgFS daemon

4 Implementation

Explanation of the implementation of ImgFS is restricted to the most important functions. We first explain the implementation details of ImgFS for image write operation, followed by the implementation for image read operation.

4.1 Write image operation

When ImgFS receives a system request to create or save an image file on the local disk, it would call up a set of functions to perform the encryption operation, as described below:

```
Class imgFile
    {if (should_enc == true)
       {img_write();
        keys_generation();
         chunks_write();
         file_encode();
         get_hpek_key();
         save_header();
     }}
```

The class 'ImgFile' is initiated with a constructer function that receives from the server 'ImgFS class': the path of the image file, flags, and file mode type arguments. By checking the flags, if the image is to be written on disk, the function 'img_write()' would be directly called up, which receives the path name of the written image file and then opens it. Then, the function 'keys_generation()' would be called up to do several tasks. The first task generates a random 16-byte Blowfish key and 8-byte file salt using 'os.urandom()'. After that, IV0 is generated using 'IV0=hmac(FEK, FSalt)', and then the stored 'HPEK' is retrieved using the function 'get_hpek(path)' from the identified location on the argument path, which is then re-hashed to obtain the original PEK.

The image file is split into a set of blocks of 4096 bytes each. In each 'write()' system call that is captured by FUSE, ImgFS immediately manages a free buffer pool and associates it to write the image blocks data. The function 'seek(header_size+offset)' would read the range of bytes in the 'offset' for the requested block up to the next block boundary and then write it with the corresponding IV into the reserved buffer using 'chunks_write(buf, offset, IV)'. However, the read operation excludes the file header length. Subsequently, the functions 'file_encode()' would be called up to perform encryption to the entire block bytes, which is then repeated until the last image block is reached. Once all blocks of the image are encrypted, the function 'save_header()' would perform encryption to FEK and FSalt using user's PEK and then generates a file access verifier by hashing of PEK. Upon successful completion, the encrypted FEK and FSalt as well as the access verifier would be appended to the beginning of the image file header.

If the file's names protection option is activated, the function 'names_encode('base64')' would be used to encode the image file's names. Finally, the

generated encrypted block would be stored into the outgoing local buffer to be returned later to FUSE and hence to the local storage disk. Fig. 4 shows the increment on the generated file header structure associated with each image file after encryption.

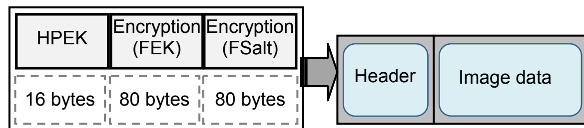

Fig. 4 Increment to the image file header after encryption

4.2 Read image operation

Similarly, when ImgFS receives a system request to read an image file from the local disk, the following functions would be called up to perform the decryption operation:

```
Class imgFile
 {if (should_dec == true)
   {img_read();
     parse_header();
     check_verifier();
     chunks_read();
     get_userkey(PrEK);
     file_decode();
  }}
```

After checking the path, flags, and mode in the system request, if they are related to an image file stored under the hierarchy tree of the home directory, the function 'img_read()' is called up to open the requested image file specified on the path argument using 'os.open(path, flags, *mode)'. Then, the function 'get_userkey(media_path, PrEK)' would be called up to load the user's PrEK from his/her removable storage device identified on the 'media_ path' argument.

Following that, the 'parse_header()' function would be called up to extract the encrypted FEK and FSalt as well as the file verifier from the image header. Here, the 'check_verifier()' function would first check the file verifier to verify that the accessed file belonged to the same user, before the read operation is executed. If the verification is realized, the user's PrEK would be used to decrypt the extracted FEK and FSalt keys. Then the function 'chunks_read(length, offset)' would be called up to read the requested block of 4096 bytes into a local buffer, and subsequently calls the 'file_decode()' function to decrypt it using the corresponding FEK and IV. Thus, the operation is repeated with all subsequent blocks and the result would be temporarily written into outgoing buffer to be returned later to the caller.

To support multiusers, we have created a 'change_user()' function which switches the current root user (the one started the FUSE server) to the normal calling user. Here, the file system is executed with the 'allow_other' option and several calls need to check (with the help of PAM) whether the requested user access is permitted, before mounting ImgFS over his/her home directory to enter a secure mount session.

5 Performance evaluation

The performance of ImgFS was evaluated by carrying out experiments using image files of different sizes. There are three objectives of the evaluation. The first objective is to compare writing and reading times of image files in ImgFS with the normal write and read in the standard Ext4 with respect to image sizes. The second objective is to perform an extensive analysis of the execution time of the major processes executed during image write and read operations on ImgFS. The third objective is to compare the performance of ImgFS with the performance of related work.

5.1 Evaluation methodology

ImgFS was tested with different image formats (compressed and non-compressed), different color depths, and pixel resolutions, and then evaluated. We ran all the experiments on a machine equipped with Intel Core 2 Duo 2.1 GHz CPU, 3 MB Cache, 4 GB main memory, and 320 GB hard disk of 7200 r/min. The machine was installed with Linux Fedora 19 of kernel version 3.12 and the Ext4 file system. Then we installed ImgFS on the same machine with three user accounts, and the option was given to run the system at the user's login time.

The experiments were carried out to compare the performance of the normal image file's write and read on the standard Ext4, against the image write with encryption and read with decryption on ImgFS, respectively. The measurements on the image file write

and read time overhead were done by sequentially splitting the image file into blocks of 4 KB each.

The experiments were performed on two groups of image files. The first group had small JPEG images of sizes starting from 25 to 1000 KB, and the second group had large BMP image files of sizes starting from 10 to 500 MB.

To ignore temporal variations, each test was repeated 15 times and the average value was taken. To ensure the accuracy of the results obtained, the file caches were flushed between each test. The computed standard deviations in all the tests were less than 5%, which indicated that the variations were not high.

5.2 Comparison with normal write and read

To measure the elapsed time of the write operation, the image file was copied to different locations under the mounted home directory, and it was copied again to the same locations after dismounting ImgFS. Accordingly, the elapsed times for all processes executed during the writing operations were recorded. Figs. 5 and 6 show the comparison on the actual time measured for writing small and large image files, respectively, without encryption using standard Ext4 versus the write time spent with encryption by sequentially writing all 4-KB image blocks using our ImgFS. The time included those of all required file system processes performed inside the kernel and at the user level during the image write operation.

From the evaluation results, it is noted that ImgFS took 72 ms, 1 s, 7.4 s, and 42.5 s to write and encrypt images of sizes 1, 20, 100, and 500 MB, respectively, whereas the standard Ext4 took 27 ms, 490 ms, 2.26 s, and 19.4 s to write the same image sizes, respectively. Hence, ImgFS could always achieve an average speed of 6.9 KB/ms and 12.9 MB/s, respectively, for writing small and large images with encryption using a block of size 4 KB, compared with the normal write using Ext4 which could achieve an average speed of 18.6 KB/ms and 36.3 MB/s for writing small and large images, respectively.

The read experiments were carried out with the same setup and image sizes as the write experiments to evaluate the performance of ImgFS against the image read operation, except that instead of writing the image file to the mounted home directory tree, the image file was read from there. Figs. 7 and 8 show the comparison on the actual time measured for reading small and large image files, respectively, with decryption using ImgFS versus the normal image files' read without decryption using the standard Ext4.

It is observed from the results that ImgFS could always achieve an average speed of 9.1 KB/ms and 39.1 MB/s on reading small and large image files, respectively, with decryption using block size 4 KB,

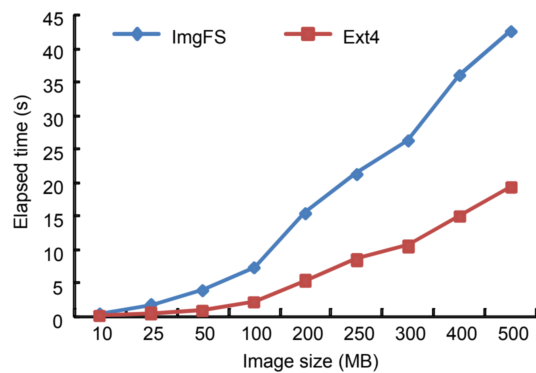

Fig. 6 Comparison of computational times for writing large BMP image files using ImgFS and Ext4

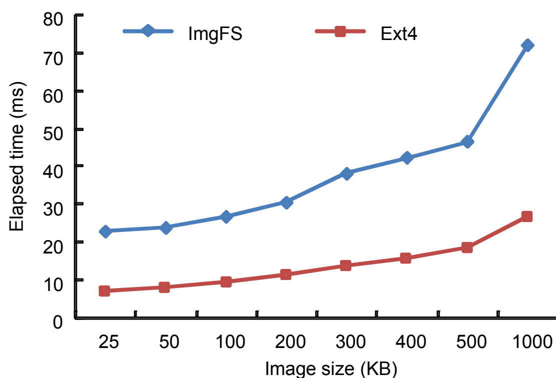

Fig. 5 Comparison of computational times for writing small JPEG images using ImgFS and Ext4

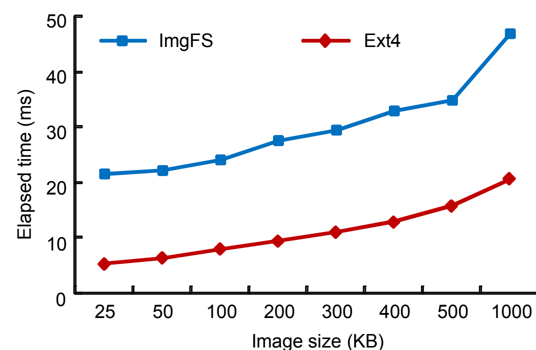

Fig. 7 Comparison of computational times for reading small JPEG images using ImgFS and Ext4

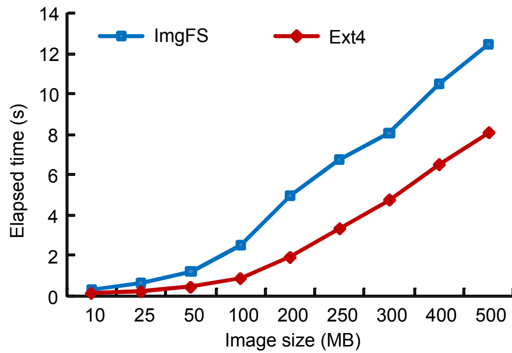

Fig. 8 Comparison of computational times for reading large BMP image files using ImgFS and Ext4

compared with the normal read on Ext4 which could achieve an average speed of 19.3 KB/ms and 90 MB/s for reading small and large image files, respectively.

5.3 Performance analysis

We measured the times elapsed by the major individual processes executed during the write operation on ImgFS. These times included: the time spent on searching image blocks of fixed sizes by setting the position of each block at the offset, the I/O write time spent by ImgFS during writing image blocks into a local buffer to perform the encryption, the actual encryption time, the workload time for loading and re-hashing the public key, and the save header time which includes encrypting other keys and saving them on the image header file.

Table 1 illustrates the measured times for those executed processes using various image sizes. Fig. 9 displays the comparison between the actual encryption time and the time spent by other write-related processes performed during the write image operation on ImgFS, namely loading PEK, save header, write seek, and I/O processes.

It is observed from the results that the save header process created a considerable performance overhead on small images, and this is because the save header process includes the operations of encrypting the file FEK and FSalt, generating the access verifier (HPEK), and then saving them all in the image file header. The overall time of those related encryption processes ranged between 41% and 94% of the total write time for the small images. Time in the large images took on average 27% of the total writing time, and the actual encryption process took the rest of the time (73%).

Table 1 Computational times obtained for the major processes performed during image write operation on ImgFS

Image size	Time (ms)				
	Load PEK	Save header	I/O write	Write seek	Actual encryption
25 KB	0.42	21.30	0.11	0.09	1.02
100 KB	0.44	21.68	0.45	0.34	3.96
500 KB	0.42	22.98	2.15	1.43	19.69
1000 KB	0.45	20.35	5.57	3.47	42.25
10 MB	0.44	21.13	52.66	34.59	440.64
100 MB	0.46	22.01	1563.09	362.95	5468.64
250 MB	0.43	21.14	4806.73	855.96	15 609.84
500 MB	0.41	22.07	10 946.06	1707.16	29 872.08

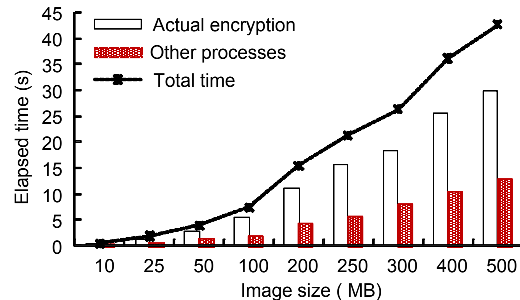

Fig. 9 Comparison of computational times for the actual encryption process compared to other write-related processes' time for writing large image files on ImgFS

Table 2 illustrates the computational times taken by the major individual processes executed during the read image operation on ImgFS, which include: loading the PrEK process; the parse header process of retrieving the stored header keys, decrypting FEK and FSalt, and then re-hashing HPEK; the read seek process of image blocks; the I/O process for reading image blocks into a local buffer to perform a decryption; and the actual decryption process.

Fig. 10 shows the computational times for the actual decryption process compared with the time taken by other read-related processes to perform the read operation on various image sizes using ImgFS.

From the results, we note that the loading PrEK created high overhead on the small images and this is because PrEK is loaded from an external storage disk which takes longer to find the right PrEK and then retrieve it. The actual decryption process of small images took an average of 22% of the total read time, whereas the actual decryption of large images created a considerable performance overhead and took about

89% of the overall read time on ImgFS while the other processes took the rest of the time (11%).

Table 2 Computational times obtained for the major processes performed during image read operation on ImgFS

Image size	Time (ms)				
	Load PrEK	Parse header	I/O read	Read seek	Actual decryption
25 KB	20.38	0.06	0.07	0.02	0.68
100 KB	21.18	0.06	0.19	0.03	2.30
500 KB	22.19	0.07	0.90	0.07	11.17
1000 KB	19.89	0.06	2.40	0.09	23.83
10 MB	20.30	0.07	17.91	0.11	235.01
100 MB	22.50	0.08	226.13	9.45	2221.33
250 MB	20.16	0.07	625.59	24.39	6048.38
500 MB	22.25	0.07	1402.24	48.93	10979.78

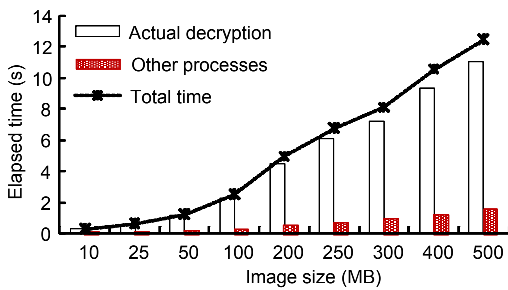

Fig. 10 Comparison of computational times for the actual decryption process compared to other read-related processes' time for reading large image files on ImgFS

We measured the processing time of the actual encryption and decryption processes on ImgFS using different block sizes. Here, we continuously read and write blocks of 1, 2, and 4 KB, respectively of large image files. Fig. 11 shows the measured times with the varied block sizes of actual image encryption and decryption processes.

We note that larger block size was more favorable for our ImgFS to exhibit its full potential. For a given block size equal to 4 KB, the encryption process was faster by about 57% and 23% than using block sizes 1 and 2 KB, respectively. On the other hand, the decryption process using block size 4 KB was faster by about 52% and 16% than using block sizes 1 and 2 KB, respectively.

The reason for this large difference in time is the extra context switches imposed by the increasing number of blocks due to the decreased block size, adding to extra overhead caused by the extra I/O, and the associated seek processes.

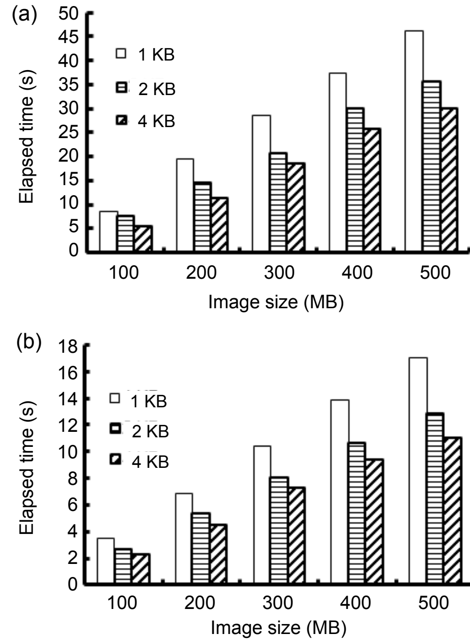

Fig. 11 Comparison of computational times for actual encryption (a) and decryption (b) processes on ImgFS using block sizes of 1, 2, and 4 KB

5.4 Comparison with the EncFS file system

We compared the computational times of write and read operations of ImgFS with one of the most popular schemes that uses a similar design approach of a transparent cryptographic FUSE-based file system with high performance, called EncFS. EncFS was installed in the same test machine with configurations similar to those of ImgFS, like the Blowfish encryption algorithm of 4096-byte block size, CFB operation mode, and 128-byte key length, an enabled option for per-file chaining IV, and random generated 8 bytes to each block header.

Fig. 12 shows the measured times of writing and reading images on ImgFS and EncFS, respectively, using various image sizes.

6 Discussion

The aim of this work is to develop a new cryptographic file system called ImgFS, which is a user-level file system based on FUSE technology and

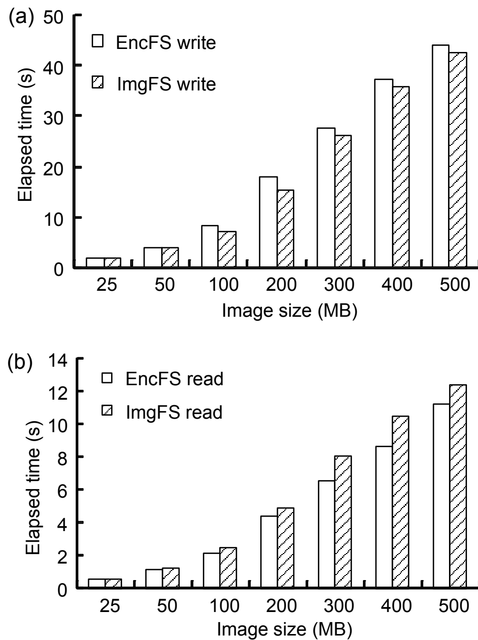

Fig. 12 Comparison of computational times for writing (a) and reading (b) images of large sizes on ImgFS and EncFS

the OpenSSL library. The development of ImgFS has also managed to overcome the identified weaknesses related to other cryptographic FUSE-based implementations:

1. ImgFS can give strong and true transparent access control to let an authenticated user enter a secure mount session during Linux login time without asking to issue mounting commands or manual passwords. This mechanism provides an enforced protection of stored images during system run time and improves its usability. This can also give ImgFS the ability to effectively work with different image applications to provide a transparent image protection service, since images produced by these applications can transparently be encrypted by ImgFS.

2. User authentication is performed using an independent authentication module that is flexible enough to authenticate individual or group users using different permissions to read or write protected images during the active session of ImgFS. The authentication module interacts with a callback handler to retrieve a user's token for PAM authentication. This authentication mechanism brings the benefits of automatic and transparent user authentication in a way completely hidden from the view of others, and it is robust enough to prevent even a super-user privilege from accessing a secure session.

3. The combination of symmetric and public key algorithms and the assigning of a unique key for each image file without storing the key in plain form on disk have addressed the weaknesses of key management. It can also be said that it is very useful from the point of view of improvement of security. This approach has also solved the sixth weakness of the current schemes that is the lack of file sharing and multi-user support.

4. ImgFS resolves the problems related to the various mounted directories under the home directory by expanding its mount to include the entire home hierarchy tree with the ability to intercept all stored image files. This expands user choices for storage without changing his/her normal habits.

5. ImgFS effectively reduces the response time from the forced decryption of all stored files on a directory each time the directory is mounted by supporting a decryption to be at the fine-grain level of an image file.

A criticism of our ImgFS implementation is that the use of a symmetric cipher may reduce the performance of the encryption process. Although several high-performance image encryption methods can be used, their security is unlikely to compete with the security levels provided by standard ciphers (Amigó *et al.*, 2007). The use of a symmetric cipher is basically a trade-off between performance and security. Moreover, by using the OpenSSL encryption library, the reliability, portability, and flexibility to support various, upgradable encryption ciphers can be easily realized. The experimental results show that even by using symmetric ciphers, the read and write performances of ImgFS can satisfy the real-time requirement.

In the evaluation of ImgFS performance, since the execution time of image write and read operations is the objective, most of the time is attributable to encryption/decryption. The cryptographic time is correlated to the extra executed context switches between the user space and the underlying kernel, which is affected by the used block size and the image file size. The block size used for encryption is limited to a maximum of 4 KB, and this limitation is due to the normal process for FUSE to break up the large writes into smaller data blocks of 4 KB each.

Therefore, this incurs extra performance overhead due to the frequently accessed data through I/O operations that go to the underlying kernel subject to the multiple cycles of encryption and decryption, whilst in the standard Ext4 no context switches were required to be performed. The various ImgFS processes analyzed through image write and read operations have performed well and run in a reasonable amount of time. This performance analysis is intended to assist in enhancing the response time of write and read operations by improving the computation time for one or more processes.

7 Conclusions

In this paper, ImgFS has been designed and implemented as a user space file system with special optimization for transparent spatial image files encryption and decryption. The primary goal of ImgFS is to have a trade-off between the high security of the storage image files and the user's convenience with a higher sustained performance. Furthermore, ImgFS has managed to solve many weaknesses in the current cryptographic FUSE-based schemes.

ImgFS has successfully achieved user convenience by supporting full transparent file system mount, user authentication, and cryptographic image files service. It can enforce strong access control to access a secure mount session as well as access protected image files by giving users the ability to perform the cryptographic task on per-image file basis, per-file keys, and per-user key-pairs. ImgFS is easily portable and applicable, and the image files can be easily shared in multi-user systems.

The performance of ImgFS has been demonstrated and the experimental results indicated that while ImgFS has managed to provide a higher level of security and transparency, its performance was competitive to other established cryptographic FUSE-based implementations of high performance. With a 4-KB block size, ImgFS is able to gain about 35.5% and 43% of the normal Ext4 time performance for writing and reading an image file with a cryptographic service, respectively.

References

Amigó, J.M., Kocarev, L., Szczepanski, J., 2007. Theory and practice of chaotic cryptography. *Phys. Lett. A*, **366**(3): 211-216. [doi:10.1016/j.physleta.2007.02.021]

Bellare, M., Canetti, R., Krawczyk, H., 1996. Message authentication using hash functions—the HMAC construction. *RSA Lab. CryptoBytes*, **2**(1):1-5.

Blaze, M., 1993. A cryptographic file system for UNIX. Proc. 1st ACM Conf. on Computer and Communications Security, p.9-16. [doi:10.1145/168588.168590]

Cattaneo, G., Catuogno, L., Sorbo, A.D., *et al.*, 2001. The design and implementation of a transparent cryptographic filesystem for UNIX. Proc. USENIX Annual Technical Conf., p.199-212.

Dowdeswell, R.C., Ioannidis, J., 2003. The CryptoGraphic disk driver. Proc. USENIX Annual Technical Conf., p.179-186.

Gough, V., 2008. EncFS Encrypted Filesystem. Available from http://www.arg0.net/encfs [Accessed on Jan. 12, 2014].

Halcrow, M.A., 2005. eCryptfs: an enterprise-class encrypted filesystem for Linux. Proc. Linux Symp., p.201-218.

Hohmann, C., 2006. CryptoFS. Available from https://github.com/reboot/cryptofs [Accessed on Jan. 26, 2014].

Jaeger, T., van Oorschot, P.C., Wurster, G., 2011. Countering unauthorized code execution on commodity kernels: a survey of common interfaces allowing kernel code modification. *Comput. Secur.*, **30**(8):571-579. [doi:10.1016/j.cose.2011.09.003]

Kerrisk, M., 2013. Linux Programmer's Manual: Kernel Random Number Source Devices. Available from http://man7.org/linux/man-pages/man4/random.4.html [Accessed on Feb. 7, 2014].

Kessler, G., 2014. File Signatures Table. Available from http://www.garykessler.net/library/file_sigs.html [Accessed on Feb. 16, 2014].

Khashan, O.A., Zin, A.M., 2013. An efficient adaptive of transparent spatial digital image encryption. Proc. 4th Int. Conf. on Electrical Engineering and Informatics, p.288-297. [doi:10.1016/j.protcy.2013.12.193]

Khashan, O.A., Zin, A.M., Sundararajan, E.A., 2014. Performance study of selective encryption in comparison to full encryption for still visual images. *J. Zhejiang Univ.-Sci. C (Comput. & Electron.)*, **15**(6):435-444. [doi:10.1631/jzus.C1300262]

Lee, K., Ewe, H., 2007. Multiple hashes of single key with passcode for multiple accounts. *J. Zhejiang Univ.-Sci. A*, **8**(8):1183-1190. [doi:10.1631/jzus.2007.A1183]

Li, S.B., Jia, X., 2010. Research and application of transparent encrypting file system based on windows kernel. Proc. Int. Conf. on Computational Intelligence and Software Engineering, p.1-4. [doi:10.1109/CISE.2010.5677091]

Ludwig, S., Kalfa, W., 2001. File system encryption with integrated user management. *ACM SIGOPS Oper. Syst. Rev.*, **35**(4):88-93. [doi:10.1145/506084.506092]

Ma, J., Li, Z., Li, J., 2010. A novel secure virtual storage device scheme. Proc. IEEE Int. Conf. on Intelligent Computing and Intelligent Systems, p.271-275. [doi:10.1109/ICICISYS.2010.5658742]

Mazières, D., 2001. A toolkit for user-level file systems. Proc. USENIX Annual Technical Conf., p.261-274.

Mellado, D., Blanco, C., Sánchez, L., *et al.*, 2010. A systematic review of security requirements engineering. *Comput. Stand. Interface*, **32**(4):153-165. [doi:10.1016/j.csi.2010.01.006]

OpenSSL Project, 2014. OpenSSL Project. Available from https://www.openssl.org/ [Accessed on Mar. 15, 2014].

Preneel, B., 2011. Modes of operation of a block cipher. *In*: van Tilborg, H.C.A., Jajodia, S. (Eds.), Encyclopaedia of Cryptography and Security. Springer US, p.789-794. [doi:10.1007/978-1-4419-5906-5_599]

Rajgarhia, A., Gehani, A., 2010. Performance and extension of user space file systems. Proc. ACM Symp. on Applied Computing, p.206-213. [doi:10.1145/1774088.1774130]

Rivest, R., 1992. The MD5 Message-Digest Algorithm. Technical Report No. RFC-1321, MIT Laboratory for Computer Science and RSA Data Security, Inc.

Schiesser, M., 2005. Complete hard disk encryption using FreeBSD's GEOM framework. Proc. 4th European BSD Conf. Available from http://events.ccc.de/congress/2005/fahrplan/attachments/586-paper_Complete_Hard_Disk_Encryption.pdf [Accessed on Feb. 9, 2014].

Shukela, V., 2013. Chaoticfs Project. Available from https://github.com/vi/chaoticfs [Accessed on Mar. 3, 2014].

Singh, V., Lakshminarasimhaiah, D., Mishra, Y., *et al*., 2006. An implementation and evaluation of online disk encryption for windows systems. Proc. 2nd Int. Conf. on Information Systems Security, p.337-348. [doi:10.1007/11961635_24]

Sunsoft, 2014. Linux-PAM. Available from http://www.linux-pam.org [Accessed on Feb. 9, 2014].

Szeredi, M., 2010. FUSE: Filesystem in Userspace. Available from http://fuse.sourceforge.net/ [Accessed on Jan. 13, 2014].

Trusted Computing Group, 2011. TPM Main Part 1: Design Principles. Specification Version 1.2, Revision 116.

Verma, O.P., Agarwal, R., Dafouti, D., *et al*., 2011. Performance analysis of data encryption algorithms. Proc. 3rd Int. Conf. on Electronics Computer Technology, p.399-403. [doi:10.1109/ICECTECH.2011.5942029]

Wright, C.P., Martino, M.C., Zadok, E., 2003. NCryptfs: a secure and convenient cryptographic file system. Proc. USENIX Annual Technical Conf., p.197-210.

Zhang, X., Liu, F., Chen, T., *et al*., 2009. Research and application of the transparent data encryption in intranet data leakage prevention. Proc. Int. Conf. on Computational Intelligence and Security, p.376-379. [doi:10.1109/CIS.2009.107]