Secure Stored Images Using Transparent Crypto Filter Driver

Osama Ahmed Khashan[1] and Nour Mahmoud Khafajah[2]
(Corresponding author: Osama A. Khashan)
College of Computing and Informatics, Saudi Electronic University[1]
Riyadh, Saudi Arabia
(Email: o.khashan@seu.edu.sa)
Computing Department, Community College, Imam Abdulrahman Bin Faisal University[2]
Dammam, Saudi Arabia
(Received May 6, 2017; revised and accepted Oct. 21, 2017)

Abstract

The threat of losing the privacy of stored images due to data breaches and malicious attacks has increased the security concerns to improve the protection of storage systems. However, the inherent features of images and the manual nature of the current encryption applications have proven to limit the prevention factors of encryption from being used more heavily in real-time. To overcome these limitations, several studies have highlighted the prominent effects of transparent encryption; nevertheless, the run-time processing in the current implementations of transparent cryptographic file systems is still limited and inefficient. In this paper, we describe the design and implementation of Crypto filter driver, a fully transparent and secure cryptographic file system for Windows platform. It can dynamically realize the processes of writing and reading file images on local disk, and transparently encrypt and decrypt them on the fly. The experiments are performed to measure the performance of the crypto filter driver over images of cryptographic service write and read. Besides the robust security level provided by the new crypto filter driver, the results showed high performance.

Keywords: File System Filter Driver; Image Encryption; Transparent Encryption

1 Introduction

The rapid technological progress in the multimedia domain leads to proliferation of huge volumes of image files to be stored on the disk drives, which in many circumstances can provide vital information. Unfortunately, the lack of security provisions for stored images may invite unwanted attackers to gain access and risk images' privacy.

Security in storage domain is a complex challenge due to the long-term requirement to storing data. Unlike transmission where the security is only needed for spontaneous time event, whereas the storage latency roughly indicates the amount of time which the attacker would need to analyse the security technique applied.

The nature of risks and threats today are increasingly getting more sophisticated, transformational, and malicious. Furthermore, the security provided by technologies such as firewall, anti-virus, intrusion detection and prevention systems are not self-protected from being attacked once an intruder gains privileges to the root security since they are all operated on the application level [16, 28].

Encryption is the most effective solution to frustrate malicious attacks and prevent inadvertent disclosure. It can effectively protect the confidentiality and integrity of stored images in the face of an intruder. Nevertheless, designing systems for storage domain using cryptographic approach is an error-prone, difficult and delicate task, and it may affect system performance if it is not properly implemented. Image encryption differs from text encryption due to the fact that images may encounter a larger space, and contains complex structures and high correlation between pixels. Therefore, encrypting and decrypting digital images involve high computational overhead and processing time, which makes it a major challenge for real-time implementations [19].

A large number of end user encryption applications are available to provide encryption for different user file types and platforms. Nevertheless, most of such encryption applications suffer from several limitations and they are always influenced by the security requirements. Performance failures occur once the encryption application is unable to meet a real-time requirement due to inadequate performance. The manual nature of applications to carry out encryption, decryption and key management. Furthermore, the routine use would increase the overhead incurred by user, which would make a user careless, or intentionally leave files in plain view. Therefore, when ex-

ecution depends on a particular software or user's direct control, it may introduce a dangerous encryption scheme.

Technology of transparent encryption is the most effective solution for storage security. However, applications must request the kernel response to perform different operations on their behalf. Inserting cryptographic service into the kernel as a basic part of the underlying file systems would offer a better functionality for transparent encryption than using any automatic user-space encryption applications. It can effectively handle huge volumes of stored data with efficient performance, high level of transparency, and it is simple to use. It also increases both the security and stability with an inability for superuser privilege to run any arbitrary code with the kernel control [11].

Transparent storage encryption can be carried out using either hardware-based or software-based approaches to perform the encryption for the entire disk data, or even a part of disk for individual files, directories, or individual partitions. Software-based encryption in this domain is more flexible and popular. Cryptographic file systems can significantly tackle the limitations related to the security and reliability by incorporating advanced encryption, authentication, and key management mechanisms. Cryptographic file system can be performed as a user space encryption layer using file system in a user space (FUSE) [7], or as a middleware layer inside the kernel [8]. It can also operate at the lower level of abstraction under the real file system either as a block device layer attached to the storage disk itself [4], or as a virtual disk driver [22,25] to provide encryption to the entire single or multiple disk's partitions.

As a part of our previous work, we have developed a transparent cryptographic file system for stored image files using FUSE technology, named ImgFS [11] for Linux platform. We also have improved the performance of the ImgFS by developing the Parallel-ImgFS [12] to overcome the cryptographic overheads and enhance the response time during image read / write operations. Parallel-ImgFS was implemented by exploiting the parallelism of the multi-core computers using block-based parallel encryption of image files. Although our implemented file systems can provide high-performance cryptographic solution; the task of reading or writing large stored image files with cryptographic service still suffers from heavy workloads due to the FUSE structure. FUSE generally suffers from a limitation of lower performance compared to other kernel file system layers. This is due to the additional overhead associated with the context switches when the FUSE passes between user space and the underlying file system [26].

In this paper, a new cryptographic file system called Crypto filter driver has been introduced for Windows platform to provide more effective transparent cryptographic service for stored images stored on disk on a per image file basis, which improves the efficiency. The developed crypto filter driver is implemented as a middleware layer inside the Windows kernel by using the file system filter driver technology. The main aim of developing this crypto driver is to attain a higher processing speed and to enhance the response time of encrypting and decrypting large image files. Moreover, to provide a systematic way to access, manage, and control all cryptography and key management operations. Therefore, we will evaluate the performance of the developed crypto filter driver over image files' read / write operations. Finally, we will compare the obtained computation results with our previous work results of ImgFS and Parallel-ImgFS versions.

The reminder of this paper is organized as follows. Section 2 presents a review of related work. Section 3 provides an overview of file system filter driver technology. Design and implementation details of the crypto filter driver are presented in Section 4. Section 5 discusses the performance evaluation. Finally, the conclusion of the paper is given in Section 6.

2 Related Works

The innovative idea of transparent services provided by the file system filter driver technology has stimulated much research in this area. Many works have been established using transparent encryption technology to provide user protection and information secrecy, and without any required change in the operating system functions. In our previous work [13] we proposed an efficient approach that can effectively trade-off between security and performance of spatial image encryption through performing a transparent partial encryption and shuffling of image blocks that was implemented inside a file system filter driver. Another proposed model by [21] for data leakage prevention. It used a double cache file system filter driver through allocating buffers in Windows kernel to manage and encrypt sensitive data that are accessed by authorized applications only. The authors of literature [34] proposed a framework model for real-time monitoring and access control to protect spatial geographical files. The filter driver focused on tracking and analyzing the copyright of protected data, and then performed transparent encryption or decryption when a user supplying the correct keys. Literatures [3,14] devoted to enhancing the security of office documents by using transparent encryption that was embedded inside the filter driver. A further proposed model for intelligent transparent encryption was discussed in [30]. It based on a filter driver to encrypt high secret level files that are evaluated and identified using a safety assessment program inside an intranet network. A prevention sensitive data leakage model proposed by [29] for transparent data encryption and real-time monitoring, which was implemented inside a filter driver for intranet environment.

Several authors have proposed models based on filter driver that play an equally important role in the field of information security as well as access control. Literature [27] proposed a transparent prevention system for illegal files access using an information flow detection algo-

rithm that was implemented inside a filter driver. A security service on cloud platform as a well established model proposed by [20] that based on a filter driver to provide transparent encryption and cloud authentication to protect and regulate virtual works. Digital rights management model was proposed by [32] using digital watermark together with a filter driver for the protection of spatial geographical files. The authors of literature [33] based on a filter driver to propose a backup model to monitor defined events belong to an application, and then back up such events or recover them when needed from the storage.

Most of these works are only proposals with less obvious implementation details, and without relative performance evaluation and testing process. Although such works might provide a satisfying solution for data security, there are still some inherent limitations in terms of transparency, flexibility, and efficiency. Current encrypting file systems provide encryption at a fine-grained level of encrypted folder or partition that include all sensitive files inside, and they do not support encryption on per specific file-type basis or specified program's file encryption. Therefore, once the file system is mounted over that folder or partition, all of its stored contents will be decrypted with extra performance overhead. On the other hand, all encryptions are performed using a single key that is stored along with data inside the local disk in plain view.

3 Overview of File System Filter Driver

File system filter driver is an optional driver tailored and attached above the file system driver inside the Windows kernel. Inside the Windows kernel space there is a set of drivers existing between user space applications and hardware devices that are grouped together in stacks and integrating with the I/O Manager. When a user threads an I/O request to open, create, read, write, or close a file, the system call request would be sent to the I/O Manager in the kernel space. In the following, the I/O Manager carries out the required processing, like parsing the filename, finding the physical location on the hard disk and creating the necessary buffers. In the end, it will build the required I/O Request Packet (IRP) before passing it down to the entry point of file system drivers. Drivers use a set of routines to handle with the IRPs through the different file system drivers' levels. Therefore, drivers read or write data from disk drivers, and then return the response back by I/O management to the user's process [31].

Windows uses a memory mapping mechanism to improve the efficiency of the file system. It maps a file into a memory space and accessing it whenever the file is needed. It uses a Fast Dispatch routine to process all fast I/O requests, and then stored data is taken out from the cache memory to the I/O Manager. Unfortunately, this leads to the difficulty of attaching any custom filter driver to capture the processes access the memory, or changing the control structure. On the other hand, Windows uses Dispatch routines to process the IRP requests that handle data obtained from disk partition by disk driver through swapping and paging activities, which are then returned to the I/O Manager [2]. This gives an opportunity to attach a filter driver to add new features, like caching, locking, compressing, security, recoverability, etc., or modify the behavior of other drivers. Therefore, whenever the IRP requests are sent to the local disk driver with a specified function call, the attached filter driver will effectively intercept these requests to carry out its task that was being designed, on the fly, before they reach the lower file system drivers. Figure 1 illustrates the structure of the file system filter driver in addition to the interaction between different kernel parts.

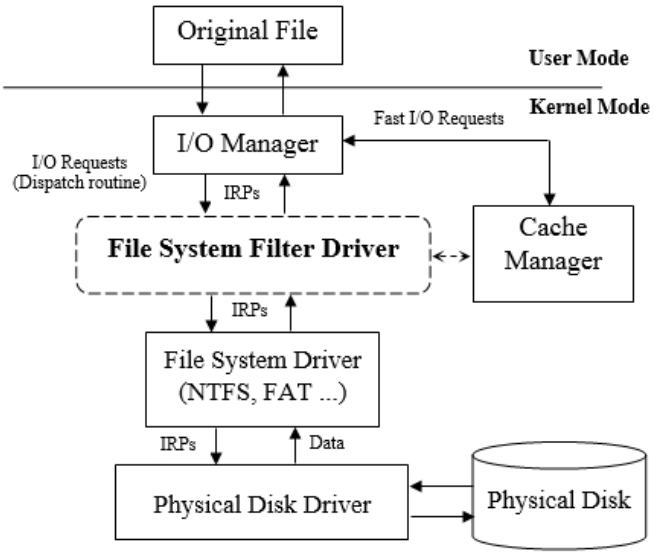

Figure 1: File system filter driver structure

4 System Design and Implementation

The overall objective of this study is to develop a crypto filter driver to provide mandatory encryption and decryption services for all image files stored on disk drive in high performance and secure execution. It has also managed to overcome the identified weaknesses related to other cryptographic filter driver based implementations.

The crypto filter driver will be able to automatically recognizing the image file once it detects that the process is trying to open or save an image file on a local disk. It will then be able to transparently encrypt or decrypt the image contents at the granularity of per image file level. All the operations will be performed in a dynamic and seamless way, neither adding overhead on the users for ciphering nor paying attention to the key management related problems.

4.1 Crypto Filter Driver Work Mode

The crypto filter driver is built on the top of the file system, therefore it can block all read and write requests related to the stored image files. When a user threads a request to write or read an image file through WriteFile() or ReadFile() system calls, the I/O Manager handles the request and builds the required IRP, and subsequently turns it directly to the underlying file system drivers. Once the IRP passes through the crypto filter driver, it exposes the logical structure of the IRP by reading the file object and header attributes to recognize whether it is for read or write an image file on a disk. With the write and read destination addresses, the crypto driver is able to encrypt or decrypt image data before it is written on disk or sent back to the user mode.

In our system, the crypto filter driver mounts itself over any file system driver and listens to all requests trying to read or write images from its associated disk partition by disk drivers. Therefore, all image files stored at the secure partition are considered confidential (automatically encrypted), and are not allowed to leak. Other files created by normal processes will not be encrypted or decrypted.

There are two methods to determine image file formats. The first method is by using the filename extension by determining the format of an image file based on the portion of the filename. However, this approach suffers from security concerns when an image format is renaming and treating as a different format, or hiding the image file extension. The other method is by using the metadata contained in the header of the file. Each file header contains a magic number that can uniquely distinguish the format of the file. Other metadata in the image header store information about image file, colour space, resolution, and other authoring information. Although this approach take longer time to identify a file format, it offers a secure way to guarantee that a file format will be identified correctly. The crypto filter driver is designed to recognize image files based on the magic numbers contained on the file header. To do this we use a list that contains most of the image magic numbers from both of well-known compressed and uncompressed image formats [10].

The pre-processing operation that is carried out by the crypto driver is to detect the image file by reading its magic number, and if that magic value is included in the list, the image file will be considered for encryption process. This makes the crypto driver most convenient to work with image applications.

Image encryption and decryption processes are carried out in the complete dispatch routines of IRP_MJ_WRITE and IRP_MJ_READ, respectively, through the completion routine set by IoSetCompletionRoutine(). When the requirements are determined for reading or writing an image file from a place on a local disk under the mount of the crypto filter driver, the image file will be considered to be added into a created encryption linked list. The File Control Block (FCB) is used to represent the confidential image files opened, and it is stored in the image file object. Consequently, the file system will generate a memory area for each image file to save image contents, regardless of how many times the image file is opened since each image has only one FCB. All opened confidential images are stored in the encryption-linked list. Therefore, when an image file is opened, the FCB pointer will be gotten from the file object that is available in the IRP. Then, the FCB pointer will be compared to an encryption file table to determine its availability. If the pointer is encountered, the FCB will be immediately added into the encryption-linked list. Confidential image file read or write is associated with adding or deleting FCB from the encryption linked list. Consequently, the corresponding image file's content will be placed into a local buffer to apply encryption or decryption operation.

4.2 Encryption and Decryption Processes

Inside our implemented crypto filter driver, there are two modules, cryptographic and key management. Figure 2 shows the implementation architecture of the image crypto filter driver.

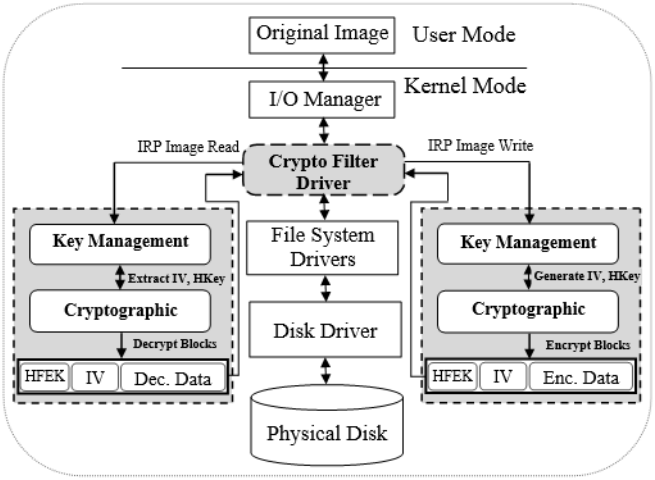

Figure 2: Crypto file system architecture

When the crypto driver realizes a new image in the list needs to be encrypted, it immediately initializes the encryption algorithm and other encryption parameters, which are realized on the cryptographic module. The whole image contents are firstly divided into a number of blocks of fixed size, each with 16 bytes. Obviously, each block should be checked that it has a perfect 16 bytes size; otherwise, the block will be padded while it is being written. Thereby, all image blocks would be encrypted sequentially one block at a time, except the image file header, which will be excluded from being encrypted or decrypted.

We picked AES as a fast symmetric encryption algorithm [5] with default 128-bit key length. Furthermore,

we chose the cipher block chaining (CBC) as an operation mode. Although the use of symmetric algorithms may provide a lower performance for the image encryption process compared with other image encryption methods, the security level provided by the symmetric ciphers is higher [15]. Thus, a trade-off between performance and security can be achieved with the usage of the symmetric ciphers.

In order to guarantee of achieving a better security level, the uniqueness requirement for the initialization vector (IV) across all image files is considered. Therefore, a unique IV of 64 bits is generated randomly each time a new image file is created. After completion of image blocks encryption, the random generated IV is attached to the header of the image file. Upon completing the encryption process, a tag is added to the head or tail of the image file. This method leads to easily identifying the encrypted image when it is read from disk during responding to the read dispatch routine in order to satisfy the decryption process.

Similarly, reading or copying stored image is performed transparently in reverse order. In the IRP_MJ_CREATE routine, the crypto driver extracts the file object from IRP stack. It then checks whether the file has an encryption tag. If the IRP is related to a stored image in a place under the crypto driver mounting point, the driver immediately responds by adding the file to the decryption list. Key management module is used firstly to extract the IV from the image file header. After that, the cryptographic module initiates the AES encryption cipher and related parameters, and then uses the associated encryption key to decipher all image blocks. The padded bytes are then removed from blocks, if exist. Once the plain image is generated, it will be directly sent back to the I/O Manager and then to the caller in the user space.

4.3 Key Management Process

The security of any cryptographic system is relied on the stringent level of the applied key management. Losing or forgetting encryption keys due to a long storage period leads to losing access to all stored data on a disk. In addition, storing keys in plain form on disk would increase the keys chances to be stolen or leaked out easily. In such study, we enforce the security of encryption keys, to overcome the key issues of authenticated encryption schemes identified by [9]. Furthermore, we allow the crypto driver to use and manage the encryption keys of all stored images in a fully transparent manner. It also addresses the limitations of current key management schemes that are operated manually on a per-file system basis.

Key management module involves the operations of creating, using and retaining the encryption keys. Using a single key to encrypt all image files is not secure, once the attacker successes to obtain the secret key for one file; he would be able to recover all other encrypted files. Therefore, in our scheme, each image is encrypted using a different file encryption key (FEK), hence, it is impossible to find two similar ciphered images related to the same plain image. We enhance the security of FEK with security margins afforded by HMAC [18], where the security analysis of HMAC is proved in [1, 17]. The FEK for each image is created by HMAC-MD5 of a common used symmetric key of 16-byte length and a hashed code of the image corresponding IV that is produced using MD5-128.

In order to ensure the integrity of the secure stored images, an embedded signature that is generated by hashing the FEK (HFEK) using MD5-128 to ensure that the image file has not been tampered or replaced by attackers during storage. The generated HFEK is then stored as an extended attribute on the header of the image file. Thus, as soon the encrypted image file is being retrieved from the disk, it would be loaded into a local buffer. The key management subsequently extracts the HFEK from the header file and the 'check_signiture()' function will check the file signature to verify that the image file has not been tampered, before the read operation is executed. When the image signature is verified, the cryptographic module will shift by 256 bits of image file header to read 16 bytes image block, and then calls the 'file_decrypt()' function to decrypt it using the corresponding FEK. The process is repeated with all subsequent image blocks and the result will be stored temporarily into a buffer in order to be returned later to the caller in the user space.

5 Performance Evaluation

In this section, several experimental tests were performed sequentially to evaluate the performance of the crypto filter driver over the write and read operations of image files with cryptographic service. A set of experimental image samples of large sizes and different formats were used. The experiment machine was installed with Windows 7 of 32-bit version, and WDK. It had an Intel Core i3- 2120, 3.3 GHz CPU. The system RAM was of size 4 GB, and the hard disk size was 320 GB of 7200 rpm.

We used Windows System Assessment Tool (WinSAT) [23] and Geekbench [6] benchmarks to run a series of tests and to evaluate the performance of the machine (in execution time) over the normal image file's write and read on the standard NTFS, against the performance of image write with encryption and read with decryption, respectively, on the implemented crypto filter driver. Each test was repeated fifteen times in each benchmark and the average of their values was taken. The standard deviation of the calculated results was not high and the intervals were always less than 6%. To ensure the accuracy of the obtained results, we flushed the cache after each test using the CcFlushCache() routine [24].

We measured the computational times of write and read operations on a number of large experimental image files using the crypto filter driver. We, respectively, wrote and read an image file of size 25 MB from places on local disk under the mount of the crypto filter driver and NTFS, and the elapsed writing and reading times were

recorded. The tests were regularly repeated by increasing the size of the image file up to 500 MB, where the sample of image files (above 250 MB) are uncompressed images of type BMP (Windows bitmap).

The recorded times include the total time for the operations of write/read image into local buffer, encrypting or decrypting image blocks, in addition to the time for extracting or saving keys on the header of image file. Figure 3 and Figure 4 show the comparison of total times measured (in seconds) for writing and reading image files using the standard NTFS and the crypto filter driver, respectively.

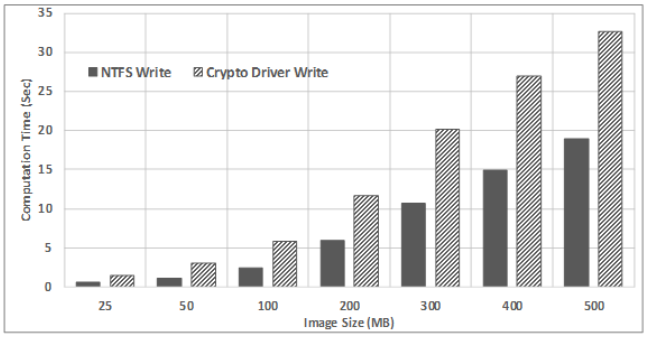

Figure 3: Comparison of computation times spent for writing images using Crypto filter driver and NTFS

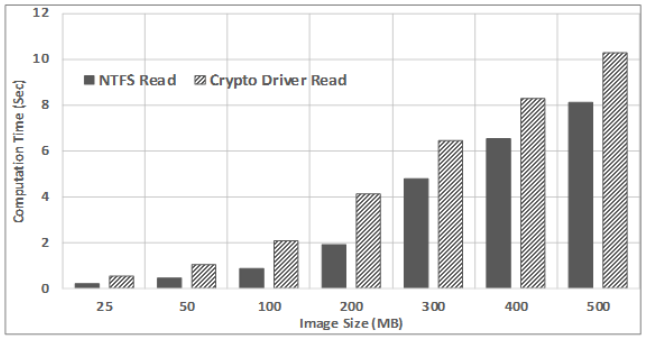

Figure 4: Comparison of computation times spent for reading images using Crypto filter driver and NTFS

From the evaluation results, it was noted that the crypto filter driver could always achieve an average speed of 16 MB/s for writing image files with encryption on the local disk, in comparison with the normal write using NTFS which could achieve the average speed of 35 MB/s. It was also observed from the results that the crypto filter driver could achieve the average speed of 48 MB/s for reading image files with decryption. Compared with the normal read on the standard NTFS, it could always achieve 91.7 MB/s as the average of reading speed.

Several major processes executed during image write and read operations in the crypto filter driver. These processes include of searching image blocks of fixed 16-byte size, writing/ reading image blocks into local buffer, the workload time for generating and loading encryption keys into/from the image file header, in addition to the actual encryption/ decryption time of all image blocks.

We measured the computational time elapsed for generating keys and saving them on the image file header during image write operation throughout the crypto filter driver mount time. It took in average 0.35 second from the image write time. We also measured the average time elapsed to load and re-generate the keys during image read operation, and it took about 0.27 second from the total image read time. Table 1 illustrates the times spent by the write-related processes and actual encryption, the read-related processes and actual decryption times elapsed during image write and read operations, respectively, in the crypto filter driver.

Table 1: Computational times elapsed for the write-, read-related processes, actual encryption, and actual decryption

Image size (MB)	Time (sec)			
	Write-related processes	Actual encrypt	Read-related processes	Actual decrypt
25	0.58	0.91	0.32	0.21
50	0.93	2.13	0.39	0.64
100	1.57	4.27	0.61	1.49
200	2.96	8.62	0.93	3.24
300	4.53	15.44	1.25	5.19
400	6.06	20.59	1.59	6.73
500	7.11	26.03	1.91	8.38

Following that, we compared the performance of the crypto filter driver over the write and read operations with our previously implemented cryptographic FUSE-based file systems, namely ImgFS and Parallel-ImgFS. Figure 5 and Figure 6 show the measured computation times for writing and reading images on ImgFS, Parallel-ImgFS and Crypto filter driver, respectively, using the same experimental image samples.

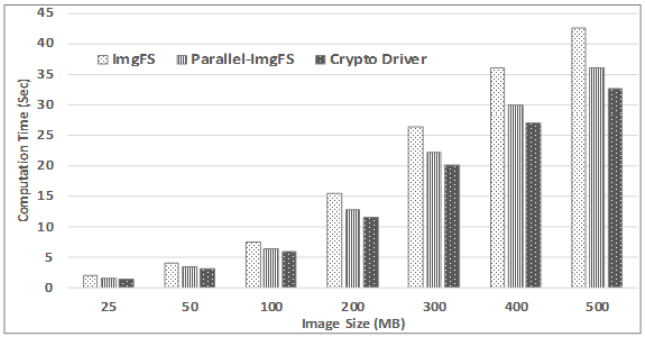

Figure 5: Comparison of computation times for writing images with encryption using ImgFS, Parallel-ImgFS, and Crypto filter driver

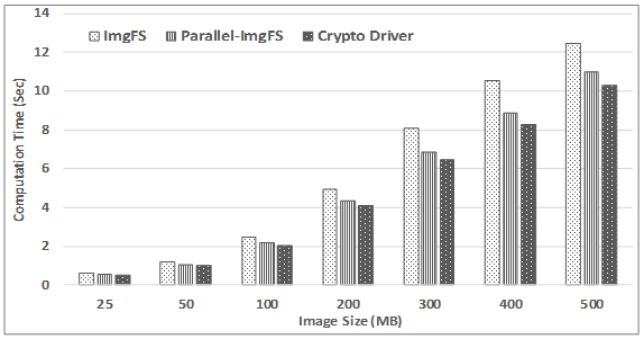

Figure 6: Comparison of computation times for reading images with decryption using ImgFS, Parallel-ImgFS, and Crypto filter driver

From the results, it was noted that the crypto filter driver could achieve higher performance (lower execution time) for both write and read operations than Parallel-ImgFS and ImgFS. When calculating the average performance, the crypto filter driver always took about 48% of the normal write time and about 57% of the normal read time for writing and reading image files, respectively. As a result, the new crypto filter driver has successfully enhanced the response time of writing and reading image files with an efficiency of 9% and 6% of the Parallel-ImgFS write and read performance, respectively.

6 Conclusion

In this paper, we designed and implemented crypto filter driver based on the file system filter driver technology that was running inside the Windows kernel. The crypto filter driver can effectively provide mandatory, automated and transparent encryption scheme for all stored images during system run time with an improved user's convenience. The system is perfectly suitable to work with image applications that might provide important information. It is more convenient to work with medical imaging systems, military image databases, scientific images, geography image sensing and personal image albums.

We have shown the implementation details of the crypto driver in order to make up the shortcomings of the existing cryptographic filter driver based implementations. It has successfully managed to reduce the response time from the forced decryption of all stored files when other cryptographic file systems are mounted, by supporting a decryption to be on per-image file basis. Key management was also perfectly suited to provide different keys for different encrypted images and without storing keys in plain form on disk.

The experimental results indicated that while the new crypto filter driver managed to provide a higher level of security, it could achieve higher processing speed with a reduced response time. It is always able to gain an average speed of 16 MB/s and 48 MB/s, respectively, for writing and reading image files with cryptographic service. In comparison with our previous FUSE-based works, the crypto filter driver could achieve higher response times of images' write and read of about 9% and 6%, respectively.

References

[1] M. Bellare, "New proofs for NMAC and HMAC: Security without collision resistance," *Journal of Cryptology*, vol. 28, no. 4, pp. 844-878, 2015.

[2] California Software Labs, *I/O File System Filter Driver for Windows NT*, Technical Report XP002548991, California Software Labs, Pleasanton, California, 2002.

[3] J, Chen, and J. Ye, "Research on the file encryption system based on minifilter driver," in *The 13th International Conference on Man-Machine-Environment System Engineering, Berlin, Heidelberg*, vol. 259, pp. 175-182, 2014.

[4] R. Dowdeswell, and J. Ioannidis, "The CryptoGraphic disk driver," in *Proceedings of the Annual USENIX Technical Conference (USENIX'03)*, pp. 179-186, 2003.

[5] D. Elminaam, H. Abdual Kader, and M. Hadhoud, "Evaluating the performance of symmetric encryption algorithms," *International Journal of Network Security*, vol. 10, no. 3, pp. 213-219, 2010.

[6] Geekbench 4, Jan. 13, 2017. (`http://geekbench.com/index.html`)

[7] V. Gough, *EncFS Encrypted Filesystem*, Jan. 27, 2017. (`http://www.arg0.net/encfs`)

[8] M. A. Halcrow, "eCryptfs: An enterprise-class encrypted filesystem for Linux," in *Proceedings of the 2005 Linux Symposium*, pp. 201-218, 2005.

[9] M. Hwang, and C. Liu, "Authenticated encryption schemes: Current status and key issues," *International Journal of Network Security*, vol. 1, no. 2, pp. 61-73, 2005.

[10] G. Kessler, *File Signatures Table*, Sep. 3, 2017. (`http://www.garykessler.net/library/file_sigs.html`)

[11] O. A. Khashan, A. M. Zin, and E. A. Sundararajan, "ImgFS: Transparent cryptographic storage images using file system in user space," *Frontiers of Information Technology & Electronic Engineering*, vol. 16, no. 1, pp. 28-42, 2015.

[12] O. A. Khashan, A. M. Zin, and E. A. Sundararajan, "An optimized parallel encryption for storing image files using filesystem in userspace," *International Journal of Advancements in Computing Technology*, vol. 6, no. 2, pp. 126-135, 2014.

[13] O. A. Khashan, and A. M. Zin, "An efficient adaptive of transparent spatial digital image encryption", in *The 4th International Conference on Electrical Engineering and Informatics (ICEEI'13)*, vol. 11, pp. 288-297, 2013.

[14] N. M. Khafajah, K. Seman, and O. A. Khashan, "Enhancing the adaptivity of encryption for storage electronic documents," *International Journal of Technical Research and Applications*, vol. 2, no. 1, pp. 28-32, 2014.

[15] O. A. Khashan, A. M. Zin, and E. A. Sundararajan, "Performance study of selective encryption in comparison to full encryption for still visual images," *Journal of Zhejiang, University Science C*, vol. 15, no. 6, pp. 435-444, 2014.

[16] S. Kim, W. Park, S. Kim, S. Ahn, and S. Han, "Integration of a cryptographic file system and access control," *Intelligence and Security Informatics, Springer, Berlin, Heidelberg*, vol. 3917, pp. 139-151, 2006.

[17] J. Kim, A. Biryukov, B. Preneel , and S. Hong, "On the security of HMAC and NMAC based on HAVAL, MD4, MD5, SHA-0 and SHA-1," in *International Conference on Security and Cryptography for Networks*, pp. 242-256, 2006.

[18] H. Krawczyk, M. Bellare, and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, RFC 2104, 1997.

[19] X. Li, C. Zhou, and N. Xu, "A secure and efficient image encryption algorithm based on dna coding and spatiotemporal chaos," *International Journal of Network Security*, vol. 20, no. 1, pp. 110-120, 2018.

[20] Z. Liang, S. Jia, J. Chen, and P. Chen, "Security of virtual working on cloud computing platform," in *IEEE Asia Pacific Cloud Computing Congress*, pp. 72-75, 2012.

[21] J. Liu, S. Chen, M. Lin, and H. Liu, "A reliable file protection system based on transparent encryption," *International Journal of Security and Its Applications*, vol. 8, no. 1, pp. 123-132, 2014.

[22] Microsoft, *BitLocker Drive Encryption Overview*, Jan. 6, 2017. (`https://technet.microsoft.com/en-us/enus/library/cc732774(v=ws.11).aspx`)

[23] Microsoft, *Windows System Assessment Tool (WinSAT'17)*, Jan. 8, 2017.(`https://technet.microsoft.com/en-us/library/cc770542(v=ws.11).aspx`)

[24] Microsoft Developer Network, *Device and Driver Technologies*, Jan. 9, 2017. (`https://msdn.microsoft.com/en-us/library/windows/hardware/ff539082(v=vs.85).aspx`)

[25] A. Patrascu, M. Togan, and V. Patriciu, "Deduplicated distributed file system using lightweight cryptography," in *IEEE International Conference on Intelligent Computer Communication and Processing (ICCP'15)*, pp.501-506, 2015.

[26] A. Suresh, G. Gibson, and G. Ganger, *Shingled Magnetic Recording for Big Data Applications*, Technical Report CMU-PDL-12-105, Parallel Data Laboratory, Carnegie Mellon University, May 2012.

[27] W. Tang, Y. Xu, G. Wang, and Y. Zhang, "An illegal indirect access prevention method in transparent computing system," in *The International Conference on Algorithms and Architectures for Parallel Processing*, vol. 9532, pp. 264-275, 2015.

[28] A. Tayal, N. Mishra and S. Sharma, "Active monitoring & postmortem forensic analysis of network threats: A survey," *International Journal of Electronics and Information Engineering*, vol. 6, no. 1, pp. 49–59, 2017.

[29] Z. Xiaosong, L. Fei, C. Ting, and L. Hua, "Research and application of the transparent data encryption in intranet data leakage prevention," in *International Conference on Computational Intelligence and Security (CIS'09)*, pp. 376-379, 2009.

[30] P. Zhang, and Z. Wei, "Application of intelligent transparent encryption model on intranet security," in *IEEE International Conference on Information Theory and Information Security*, pp. 268-270, 2010.

[31] C. Zhang, Y. Wu, Z. Yu, and Z. Li, "Research and Implementation of File Security Mechanisms Based on File System Filter Driver," in *IEEE Annual Reliability and Maintainability Symposium*, 2017.

[32] L. Zheng, L. Feng, Y. Li, and X. Cheng, "Research on digital rights management model for spatial data files," in *The 2nd International Conference on Information Engineering and Computer Science*, pp. 1-4, 2010.

[33] Z. Zhongmeng, and Y. Hangtian, "A data backup method based on file system filter driver," in *The 2nd World Congress on Software Engineering (WCSE'10)*, vol. 2, pp. 283-286, 2010.

[34] G. Zhu, Z. Liangchen, L. Guonian, and Z. Liangchen, "The access control technology of spatial data files based on file system filter driver," in *The 11th IEEE International Conference on Communication Technology (ICCT'08)*, pp. 734-737, 2008.

Biography

Osama Khashan received his B.S degree in Computer Science from Irbid National University, Jordan in 2005, M.S in Information Technology from University Utara Malaysia in 2008, and the Ph.D in Computer Science from the National University of Malaysia in 2014. He is currently an assistant professor in the College of Computing and Informatics, Saudi Electronic University, KSA. His research works focus on information and network security, digital image processing, and performance analysis.

Nour Khafajah received her B.S degree in Computer Science from Al-Balqa Applied University, Jordan in 2011, and the M.S in Information Security and assurance from the Islamic Science University of Malaysia in 2014. She is currently a lecturer in Imam Abdulrahman Bin Faisal University, KSA. Her research intrests in information and cyber security.