cryptography

Article

Dynamic Multimedia Encryption Using a Parallel File System Based on Multi-Core Processors

Osama A. Khashan [1,*], Nour M. Khafajah [2], Waleed Alomoush [3], Mohammad Alshinwan [4], Sultan Alamri [5], Samer Atawneh [5] and Mutasem K. Alsmadi [6]

[1] Research and Innovation Centers, Rabdan Academy, Abu Dhabi P.O. Box 114646, United Arab Emirates
[2] Department of Cyber Security, Faculty of Science and Information Technology, Irbid National University, Irbid 21110, Jordan
[3] School of Information Technology, Skyline University College, Sharjah P.O. Box 1797, United Arab Emirates
[4] Faculty of Information Technology, Applied Science Private University, Amman 11931, Jordan
[5] College of Computing and Informatics, Saudi Electronic University, Riyadh 13316, Saudi Arabia
[6] Department of MIS, College of Applied Studies and Community Services, Imam Abdulrahman Bin Faisal University, Dammam P.O. Box 1982, Saudi Arabia
* Correspondence: okhashan@ra.ac.ae

Abstract: Securing multimedia data on disk drives is a major concern because of their rapidly increasing volumes over time, as well as the prevalence of security and privacy problems. Existing cryptographic schemes have high computational costs and slow response speeds. They also suffer from limited flexibility and usability from the user side, owing to continuous routine interactions. Dynamic encryption file systems can mitigate the negative effects of conventional encryption applications by automatically handling all encryption operations with minimal user input and a higher security level. However, most state-of-the-art cryptographic file systems do not provide the desired performance because their architectural design does not consider the unique features of multimedia data or the vulnerabilities related to key management and multi-user file sharing. The recent move towards multi-core processor architecture has created an effective solution for reducing the computational cost and maximizing the performance. In this paper, we developed a parallel FUSE-based encryption file system called ParallelFS for storing multimedia files on a disk. The developed file system exploits the parallelism of multi-core processors and implements a hybrid encryption method for symmetric and asymmetric ciphers. Usability is significantly enhanced by performing encryption, decryption, and key management in a manner that is fully dynamic and transparent to users. Experiments show that the developed ParallelFS improves the reading and writing performances of multimedia files by approximately 35% and 22%, respectively, over the schemes using normal sequential encryption processing.

Keywords: parallel encryption; multimedia encryption; file system; multi-core processors; cryptography

check for updates

Citation: Khashan, O.A.; Khafajah, N.M.; Alomoush, W.; Alshinwan, M.; Alamri, S.; Atawneh, S.; Alsmadi, M.K. Dynamic Multimedia Encryption Using a Parallel File System Based on Multi-Core Processors. *Cryptography* **2023**, *7*, 12. https://doi.org/10.3390/cryptography7010012

Academic Editor: Josef Pieprzyk

Received: 21 January 2023
Revised: 1 March 2023
Accepted: 2 March 2023
Published: 6 March 2023

1. Introduction

Multimedia data such as images and videos are currently playing a significant role in society, and huge volumes of multimedia files are stored on disk drives and removable storage media. Unfortunately, many potential threats and security attacks are directed towards violating the privacy of personal information on these storage devices, particularly multimedia, which prompts a significant amount of interest from the research community.

Storage encryption is the most effective solution for providing advanced protection against threats and preserving the confidentiality and privacy of storage [1]. However, designing cryptographic techniques for multimedia data at rest is complex. This is because cryptography is known to be a mathematically heavy operation, particularly for multimedia data, which have unique properties such as bulk data capacity, a high redundancy, a strong correlation between data elements, and the use of various formats for file storage, as well

as long computing times and vast amounts of processing power during encryption and decryption. This poses a great challenge for multimedia cryptosystems by preventing them from being used heavily in real time [2,3]. Other problems related to the security level provided by multimedia applications may be reasonable from the designer's point of view but do not consider sudden unexpected attacks [4]. Although existing end-user encryption applications are ubiquitous, they still suffer from several inherent weaknesses in terms of security, flexibility, transparency, and performance efficiency. The manual nature of encryption applications and the additional overhead incurred by users in performing cryptographic operations are cumbersome and time-consuming. This routine usage leads users to be careless regarding potential threats and leave files in plain text format [5].

Current research trends in cryptographic file systems address the limitations of encryption applications by implementing a dynamic mechanism for managing, controlling, and monitoring encryption, decryption, and key management operations with the help of operating system file systems. Moreover, they can perform cryptographic operations in a highly secure, transparent, and efficient manner. Cryptographic file systems exist in two forms, either implemented inside the kernel space or as a file system residing in the user space. At the kernel space level, the cryptographic file system can be implemented as a middleware layer to encrypt individual files or directories using file system filter driver technology in the Windows kernel [1,4,6,7] and Unix-like stackable file system [8,9]. Furthermore, a cryptographic file system can be implemented as a low-level file system layer, operating under the real file system, either as a block device layer attached to the storage disk itself [10] or a virtual disk driver [11] providing encryption for all single- or multiple-disk partitions.

The file system in user space (FUSE) is a Unix-like framework that allows a non-privileged user to implement a file system to perform a particular functionality in the user space [12]. Such an approach can provide a robust solution for storage security at a high transparency level without the requirement for editing the underlying kernel level or significantly changing the design and implementation of the basic file system [13]. A considerable number of studies and projects have employed FUSE technology to provide transparent encryption for various types of data at rest. In this study, we present a parallel user-level encrypted file system called ParallelFS based on FUSE technology for multimedia data. The motivations behind this study are to mitigate or completely eliminate issues related to the security, efficiency, transparency, and user convenience of existing kernel-level and FUSE-based encryption file systems. ParallelFS is designed with the main goal of providing higher cryptographic performance and achieving the lowest possible response time when reading or writing stored multimedia files. The contributions of this study include the following:

1. A parallel and fully dynamic cryptographic file system that utilizes the parallelism of multi-core processors was developed to accelerate cryptographic operations and key management and improve the system response time for stored multimedia files.
2. Hybrid methods constructed from symmetric and asymmetric ciphers and hash algorithms are used to improve the security of multimedia files and allow for efficient file sharing among multiple users.
3. The performance is analyzed, and the results are compared with those of related work. The results demonstrate the efficiency of the developed parallel file system in improving the encryption speed and shortening the system response time, as well as its ability to ensure protective security.

The remainder of this paper is organized as follows. Section 2 overviews the FUSE and discusses the design goals. Section 3 presents the related work. Section 4 details the proposed parallel-encryption file system. Section 5 discusses the performance evaluation of the implemented file system. Finally, Section 6 concludes the paper.

2. Background and Design Goals

In this section, we first provide an overview of the FUSE structure. Then, we discuss the basic design goals of the proposed scheme.

2.1. Overview of FUSE

FUSE is a user space file system development framework that utilizes the Linux high-level application interface (API) to extend the kernel-level functionality in user space. The design structure of FUSE includes the FUSE kernel driver which interacts with the kernel on behalf of the non-privileged user/application and deals with API to present a virtual device that is able to communicate with a user file system code running in the user space [14].

Generally, a file system is a type of kernel module that has two functions. The first is to provide the user with a method for accessing their data, and the second is to build a logical structure for managing the data, where both functions are realized in the kernel space. In the case of FUSE, it simply provides hooks for user space applications to provide the functionality that a kernel module would have provided to instantiate a file system, whereas the developer is responsible for realizing the logical relationship structure of the file system that is used to manage the data [15]. Logically, the FUSE is divided into a set of file system callback interfaces that are used to process incoming requests from the kernel and into a set of forward processes used to control the operations of the FUSE itself. The user is initially required to mount the FUSE over a special directory called a mount point. This mounting operation basically creates a map between the source directory and the mount point to provide a dynamically transparent service that is realized at the mount point. When FUSE is mounted, the file system type is directly registered to the Linux kernel in order to let it know that the file system on that specific directory is FUSE [16]. The mount utility is included in the FUSE library, and during the mount time, the user space file system daemon can realize the logical relationship to manage the files of the source directory and read their metadata [13]. When a system call is issued to read a file from a mount point, the virtual file system kernel (VFS) will first check whether the file is available in the kernel-space page cache to return immediately; otherwise, the system call will be forwarded to the FUSE library to invoke the callback in the user space file system daemon [17]. While the file system is running at the mount time, the FUSE daemon cannot be called by VFS without the use of FUSE. In some cases, FUSE may take another action, such as returning the requested data into the buffer, or it may perform some pre-processing by requesting data from the underlying file systems [18]. Eventually, when the mounting connection is no longer needed, the mount point directory contents automatically disappear as soon as the file system is unmounted.

2.2. Design Goals

In this work, we aim to design a cryptographic file system that dynamically encrypts and decrypts multimedia files stored on disk drives and to achieve the following goals:

1. The designed ParallelFS cryptographic file system must encrypt multimedia files at the user space before being stored on a disk so that the files remain encrypted during storage using robust encryption algorithms and strong encryption keys. Meanwhile, the designed ParallelFS should automatically decrypt the files when they are accessed by the data owner.
2. The designed ParallelFS should achieve high efficiency when performing the encryption and decryption operations, without affecting the system performance and data access response time. Moreover, the cryptographic and key management operations are realized at the fine-grained level of individual multimedia files; there is thus no need to incur additional computational costs and time delays caused by encrypting and decrypting the entire multimedia files each time the file system is mounted.
3. The designed ParallelFS must dynamically perform the encryption, decryption, and key management operations in a fully transparent manner so that the user will not

notice any difference when files are stored or accessed. In the meantime, the efforts of a user in managing the encryption keys should be reduced.

3. Related Work

Several state-of-the-art file system architectures have been proposed to perform multiple functionalities, such as encryption [1], detection [19], compression [20], and access control [6]. Kernel-level encryption file systems are designed to provide transparent encryption at the granularity of a single file [2], single partition [21], or on-the-fly full disk encrypt [22]. However, developing a file system and inserting it into the operating system kernel as a middleware or block device layer to fully function, such as for transparent encryption, is difficult and complicated, depending on many operating system specifics and interacting components of the kernel data structure [15].

Therefore, user space cryptographic file systems have been developed to provide similar features offered by kernel-level cryptographic file systems, but with less implementation effort. The authors in [23] proposed CFS as a network file system that allows an authenticated user to create a source directory in a local or remote file system to store encrypted files. The CFS daemon allows the source directory at the mount time to open a directory called the mount point, which displays the user files in an unencrypted format. In [24], the authors presented EncFS as a cryptographic file system layer in the user space, developed using the FUSE library. EncFS can be mounted over a secure directory used to store the user's sensitive files and then performs transparent encryption for all stored files using standard ciphers, such as AES and Blowfish. In EncFS, a global password is used to authenticate user access, and a single encryption key is used to encrypt all files. In [25] a working extension was provided to EncFS to enable it to support multi-user file sharing and file-based access control. Here, a trusted shared server was used to store shared files, and a key management model was designed to authenticate users for securely accessing shared files. The authors in [17] introduced ImgFS as a user-level file system built on top of FUSE to provide transparent encryption for digital images stored on a disk. In ImgFS, user authentication and access control for a secure mount session are performed through a Linux-based pluggable authentication module (PAM), which dynamically authenticates a legitimate user at the mount time for entering a secure ImgFS session, with the ability to support image file sharing between users. Furthermore, the system automatically mounts the ImgFS during the Linux login session time to transparently encrypt or decrypt the image file in its corresponding place in the original source home directory located under the root file system. The authors in [26] presented SafeFS, an FUSE-based modular architecture that provides encryption, replication, and coding features for user files in stackable building blocks and can access remote data stores. SafeFS is also designed by allowing users to customize their data store according to their specific needs by defining a set of blocks for protection and sped-up performance. In [13], the OutFS is presented, which is a cryptographic file system that provides transparent encryption for outsourced files stored on a cloud server. The file system was designed to be mounted on top of the synchronized cloud directory that stores encrypted user files and provides a virtual mount point to display the user files in an unencrypted format on the user's machine. Moreover, the developed OutFS file system handles the identity of file owners, the integrity of outsourced files, and the shareability of files among users.

However, all these schemes were designed based on the normal execution of the FUSE library, which suffers from high performance overhead owing to the effect of context switches caused by file system calls [27]. The performance and resource utilization of FUSE over different workloads have been extensively studied [28]. In [29], the efficiency of the encryption process in FUSE was improved by reducing the file decryption operations. A cache module was attached to the FUSE library to store the plaintext of the encrypted files to be accessed upon user request. Otherwise, FUSE reads the file's data from its original location on the disk and then decrypts and caches it to be reached on future system requests. Numerous parallel file systems designed to provide different functionalities are classified

into commercial parallel file systems that provide high-performance processing for I/O-intensive applications, such as GPFS for IBM [30]. They can also be used to research parallel file systems, such as PVFS [31]. Nonetheless, most parallel file systems have been developed for distributed computing to provide user machines with concurrent distributed access and processing regarding files [32]. Moreover, existing FUSE-based cryptographic schemes use sequential processing construction and do not keep pace with multi-core computing capabilities. Therefore, the multimedia encryption performance must be accelerated, and the system response time must be improved.

4. Proposed Parallel Multimedia Encryption File System

This section provides an overview of the proposed ParallelFS encryption file system. Subsequently, we present the structural details of parallel encryption using multi-core processors. Finally, the implemented hybrid encryption scheme for multimedia data is described.

4.1. Design Overview

The proposed ParallelFS was designed as a backend file system layer located in the user space to provide a transparent cryptographic service for multimedia files on the fly. This allows users to use it in a similar manner to traditional file systems with the ability to work with single- and multi-user systems. General FUSE framework is structured from the FUSE kernel driver and user space file system daemon. Through the FUSE library, the FUSE kernel driver provides the developer with a set of standard system calls that enable the development of a custom FUSE file system for adding a new feature or improving existing functionalities. Figure 1 shows the general architecture of the FUSE framework and the interactions between its components. ParallelFS was designed to interact with standard file system calls such as open, read, write, and save for multimedia files stored on a disk.

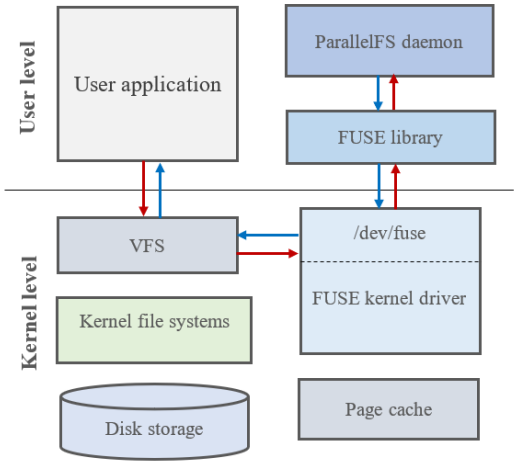


Figure 1. FUSE framework architecture.

When a user or application initiates a system call for a multimedia file, the system call is dynamically intercepted by the VFS layer of the Linux kernel. The VFS is a kernel software layer that provides an interface for all file systems and storage devices. It handles all system calls, abstracts the functionality of the file system, consults the mounted file system table, and parses the file path. When the VFS realizes that the system call concerns a multimedia file stored in a directory within the ParallelFS mount, it forwards the system call to dev/FUSE. At the user level, the FUSE library handles the main functions responsible for mounting the ParallelFS, initializes the data structure, and manages the communication between the FUSE kernel driver and ParallelFS daemon. Once the FUSE library realizes that a system call is currently in the kernel queue, it invokes the request from dev/FUSE, processes it, and involves the callback functions required to execute ParallelFS. After the

cryptographic function is executed, the FUSE library writes the results back to dev/FUSE and then to the FUSE kernel driver. Finally, the FUSE driver returns the response to either the disk for storage or the user application that sent the request.

The proposed ParallelFS file system was designed to provide a mandatory mechanism using a hybrid encryption scheme for encrypting and decrypting a multimedia file each time a user sends a request and before a write or read operation is conducted. When a multimedia file is stored for the first time in a directory under the ParallelFS mount, encryption keys are randomly generated, and the multimedia file blocks are encrypted to generate an encrypted file version in this secure source directory, without user intervention. The designed file system can support a wide range of multimedia file formats compatible with various multimedia applications, including images, audio, and video file formats. We used the magic signature stored on the metadata of the multimedia file header to recognize the file type. Moreover, the file system was designed to avoid unnecessary decryptions and re-encryptions of unused files each time a user mounts the file system by restricting the decryption process to the fine-grain level of an individual multimedia file. When a user mounts ParallelFS over a root directory, e.g., /Multidir, the hierarchy tree of this root directory becomes a mount point that automatically displays the user's selected file in an unencrypted format. This provides the user with the flexibility to store secure multimedia files without being restricted to a single directory location. Simultaneously, all stored files are transparently encrypted on the corresponding source directory, which is provided a suffix extension /Multidir.sec to distinguish it.

ParallelFS was also designed to handle a dynamic key management process and enforce user authentication during the mount time. The system administrator is responsible for installing the ParallelFS file system and configuring the authentication policy to allow a non-privileged user to mount the file system and enter a secure mounting session. Each user has a login authentication key that is used to mount the file system, which is generated from a hash of the Linux login passphrase using SHAKE-128 [33]. In addition, each user has a public and private key pair (Pk, Prk) that is used to encrypt/decrypt symmetric file encryption keys. When file data are encrypted with a symmetric key (K), the Pk of the user is used to encrypt K and append it to the header of the multimedia file. Figure 2 summarizes the interaction between the proposed multimedia file system components in performing the encryption/decryption processes for a stored/opened multimedia file.

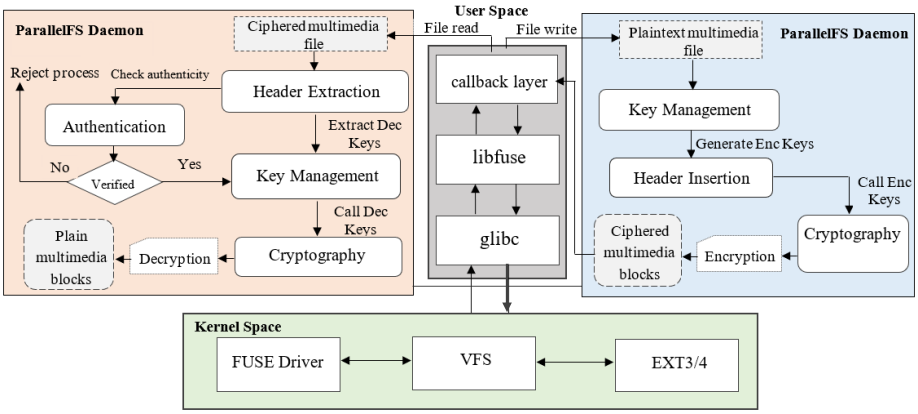


Figure 2. Workflow of ParallelFS components for multimedia file encryption/decryption.

4.2. Sequential vs. Parallel Processing Patterns

In sequential processing, encryption is composed of a chain of sequential processes of file blocks that run dependently; the encryption of each block depends on the instant output of a previous ciphered block. Therefore, reading or writing a large storage file using cryptographic services is a major bottleneck [32]. The difficulty of this challenge increases if applied to user space applications. Cryptographic file systems typically optimize performance because they combine encryption and integrity protection techniques,

and all related computations are performed in a highly seamless and compatible manner, without many data copies between the kernel and user space. However, the file system schemes were designed in such a way to primarily handle a file as an addressable sequence of bytes and blocks, thereby preventing them from exploiting the recent advances in multi-core processors.

Parallel cryptographic file systems can significantly address the processing overhead incurred by cryptographic operations and reduce the system response time. Here, the encryption and decryption of separate file blocks were computed independently and processed concurrently using multiple processes and threads in a parity form. Although cryptographic file systems require less memory in sequential construction, the parallel cryptographic file system effectively provides a high-performance cryptographic solution to treat large data files and reduce the effects of bottlenecks (imposed by heavy workloads), and it can thus be used to satisfy real-time demands [34]. We improved the performance of existing cryptographic file systems by developing ParallelFS to support the cryptographic workload by concurrently processing a common set of multimedia file blocks.

The primary goal of ParallelFS is to achieve both a higher cryptographic performance and a faster response time for each multimedia file read and write request. However, the performance level is driven by reserved processes, resource workloads, and the technological environment. Moreover, the variation in synchronization and the difference in the computation of parallel threads affect the performance of CPU cores. In a multi-core CPU, one core may be overloaded by waiting for I/O operations or entering a low-power idle state. When executing tasks that vary in size, one core may complete its process before other cores, thereby reducing the efficiency and increasing the system response time. To avoid this, in the proposed scheme, we measured the CPU utilization of threads by measuring the amount of time each thread spends executing on the core. Thread scheduling in ParallelFS is based on the completely fair scheduler (CFS) of Linux. It periodically and dynamically measures the CPU utilization of running threads such that fewer cryptographic processing tasks are assigned to heavily loaded cores, whereas larger tasks are scheduled to cores with less data processing or are in an idle state. We designed a thread pool in the ParallelFS file system daemon to perform parallel encryption tasks running on the CPU cores. Figure 3 shows the scheduling of threads on CPU cores, where Tz denotes the thread size in bytes, and U is the core utilization ratio.

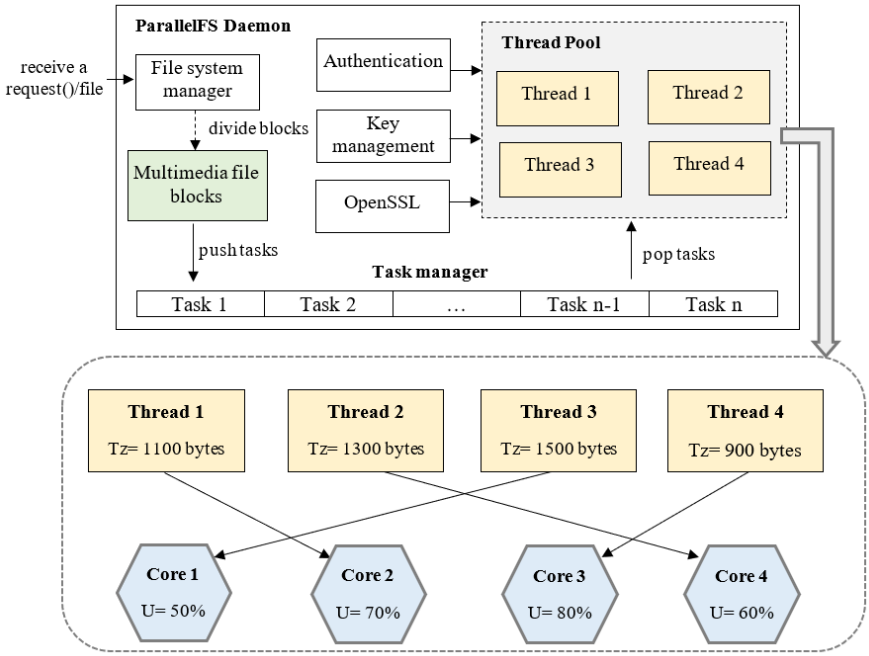
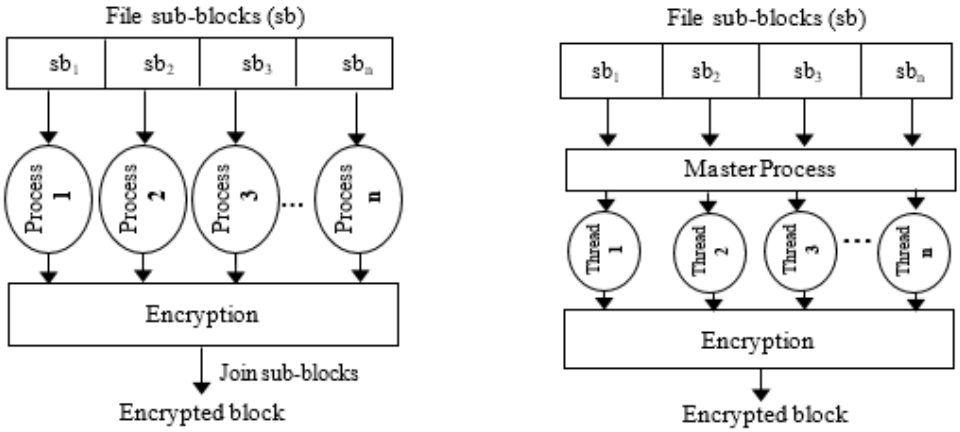


Figure 3. Structure of thread scheduling on CPU cores in the designed ParallelFS.

4.3. ParallelFS File System Structure

Parallel encryption based on a multi-core processor can be implemented using two different methods. The first method can be carried out using the forking approach by creating a number of child processes, where each child process has a different process ID and a separate memory location in a virtual memory with a different address space, which are executed independently of each other. In the second method, a threading approach can be used by creating a number of threads that belong to a single parent process and share the same address space and parameters through global variables. Figure 4 shows the architecture of parallel encryption design using fork- and thread-based parallelism methods.



(**a**) Fork-based parallelism (**b**) Thread-based parallelism

Figure 4. Parallel encryption architecture using fork- and thread-based parallelism methods.

The structure of the ParallelFS was designed using the threading approach. Here, the FUSE driver was used to hook the system calls related to the ParallelFS operations. When the I/O request accesses a file in a mount point directory, the request is forwarded to perform the customized cryptographic procedure. Otherwise, the process passes the request to the underlying kernel file system. When the *write ()* request is received, the plaintext file is divided into several blocks, each with a maximum size of 4 KB. The blocks are then split into several sub-blocks, with four sub-blocks of 1 KB each. We used a pre-fork technique by creating a task manager, and the inter-process communication (IPC) method was used to push the tasks to the task manager queue. This can provide an efficient mechanism for communication between multiple processes and reduce the effect of bottlenecks imposed by the allocation of processes and threads, which should lead to a higher performance. Moreover, we built a thread pool that includes several threads belonging to one parent process and shares the same address space and parameters through global variables. The created threads then pop the tasks from the task manager and distribute them according to the CFS CPU scheduler. Here, the push operation is given a higher priority than that of the pop operation in the task manager. In addition, a lock was designed to block threads when the task manager empties.

After the threads are created, each thread is encrypted independently, and all threads are encrypted in parallel. In addition, all encryption parameters required by the encryption function, such as the encryption key and parameters, are passed. Subsequently, the output of each thread is collected in reverse order, considering the task order, as tasks do not end in the same order. Algorithm 1 describes the steps involved in ParallelFS when a *write ()* request is received for a multimedia file.

Algorithm 1: Multimedia file write in ParallelFS

```
//Receive file write ( ) request
Begin
    (B_1, ... , B_n) ← Split (F)
    (SB_1, ... , SB_n) ← Split (B_i)
    (SB_1, ... , SB_n) → Call CreateTasks ( )
    (Tsk_1, ... , Tsk_n) → Push (Task_Manager)
    ThreadPool ← Call CreateThread ( )
    for ∀Tsk_i in Task Manager do
            Tsk_i ← Pop (Task_Manager)
            Connect Tsk_i to Thrd_x
            Create MemorySlot ( )
    end for
    //Perform parallel encryption for created threads
    for ∀ Thrd_x in ThreadPool do
            Call KeyGenration ( )
            Call ParallelEncrypt (Thrd_1,.., Thrd_n )
            Call chunks_write ( )
        Call save_header ( )
    end for
 End
```

At the beginning of this study, we attempted to pass the sub-block data as a usual parameter used in various programming languages. This implies serializing the parameter and sending it through an internal queue structure. However, we determined that this method suffers from high workloads and is considerably time-consuming. To avoid these drawbacks, we attempted to allocate shared memory before creating the process pool.

We allocated many shared memory slots, each of the same size as the sub-block, and the number of slots is that of the pre-forked processes, which should be a multiple of four with a minimum of four slots, because the structure of most present multi-core processors has four processes and threads. Simultaneously, a lock is used around each slot to ensure that all slots belong to the same task and to avoid any deadlock caused by having slots belonging to different tasks and no free slots for each to be realized. After filling the slots with the data segments, we queue the tasks for each passing key and IV, which could be in a shared memory. Moreover, for the index of each shared slot, we associated an IPC event with all service subprocesses. We then wait until the slots notify us using *event.wait()* at one end and *event.notify()* at the other. When all segments are complete and notify, the waiting caller collects the result from the same slot as the encoding function. It obtains the sub-block from a shared memory, encrypts it, and then places it back in the same shared memory. When threads of the second method are used, instead of the first method's processes, slots of global variables are used in the same manner.

4.4. Parallel Encryption and Decryption

A multimedia file written in a mounted directory is intercepted by ParallelFS. The file is then encrypted using a hybrid encryption scheme with symmetric and asymmetric ciphers. In symmetric encryption, the Blowfish encryption algorithm is used to encrypt all file blocks with a key length of 128 bits and a block size of 4 KB. A 64-bit file salt was randomly generated for each new multimedia file encryption. The counter mode (CTR) was used in our scheme. The CTR is a fully parallelizable mode of operation that works effectively on multimedia encryption and provides random access to any block cipher without error propagation or ciphertext expansion. Each data block has a unique IV generated by XORing the file salt, with the counter block corresponding to each data block. This prevents similar plaintext blocks from being encrypted to the same ciphertext block. Therefore, the uniqueness requirement of the counter block across all file blocks is necessary to guarantee greater protection. After the file body is symmetrically encrypted, the encryption key (with

a file salt) is asymmetrically encrypted using the RSA-2048 algorithm with the user's public key, which is then stored with the header of the multimedia file. Algorithm 2 describes the file encryption steps involved in the ParallelFS daemon. First, the system generates all unique encryption block counters (*ctr*) with the same number (*n*) of data blocks (*B*) in the multimedia file (*F*). Then, all unique IVs associated with all data blocks are created by XORing the global file salt (*FSalt*) with each corresponding block encryption counter (ctr_i).

Algorithm 2: Parallel multimedia file encryption

Input: *F,Pk*
Output: *CF, CK*
Begin
K, FSalt ← *RandomGenerate ( )*
B_i ← *Call CreateFileBlocks* (*F*)
SB_i ← *Call CreateSubBlocks* (*B*)
//Generate file counters
for $\forall SB_i \in (B_1, .., B_n)$ **do**
$ctr := (ctr_1, \ldots, ctr_n)$ ← *RandomGenerate ()*
end for
//Parallel sub-blocks encryption using CPU cores
for $\forall B_i \in F$ **do**
for $\forall SB_i \in B_i$ **do**
$IV_i \leftarrow FSalt \oplus ctr_i$
Connect SB_i *to* $Thrd_x$
end for
CB_i ← *ParallelEncrypt* (*Blowfish* (SB_i, *K*, IV_i))
end for
$CF := (CB_1, \ldots, CB_n)$
//Symmetric key encryption using RSA-2048
CK ← *Encrypt* (*RSA* (*K*, *FSalt*))
Return *CF*, *CK*
End

Figure 5 presents the parallel encryption processing of a data block. As the input, the encryption function takes a single 4 KB data block (B_i) segmented into *m* sub-blocks (SB_1, SB_2, ... , SB_m), the corresponding unique IV_i, and the encryption key *K*. The same IV is used, with all sub-blocks belonging to the same parent block, and the same *K* is shared among all file blocks. The parallel encryption function is ready for execution as soon as it completes receiving the encryption parameters. Each sub-block of data is associated with a thread to encrypt the data segment independently without relation to other sub-blocks, and all threads execute concurrently in parallel. Once the parallel execution of all sub-blocks is accomplished, this results in encrypted sub-blocks (CSB_1, CSB_2, ... , CSB_m), which are combined into a ciphered block (*CB*). This operation is repeated with all file blocks (*n*), and all resulting ciphered blocks are collected (CB_1, CB_2, ... , CB_n) and written to the disk as a ciphered multimedia file (*CF*).

Decryption works similarly, but in reverse. First, both *K* and *FSalt* are extracted from the image header, decrypting them using the user's private key and reconstructing all of the used IVs. As the input, the decryption function takes the encrypted blocks (CB_1, CB_2, ... , CB_n), the corresponding IV_i, and the same encryption *K*. Subsequently, each encrypted block is segmented into sub-blocks and processed concurrently in parallel. This operation is repeated until all ciphered blocks are decrypted resulting in the original plaintext of the multimedia file *F*. Algorithm 3 describes the parallel decryption steps involved in ParallelFS when a *read ()* request is received for a stored multimedia file.

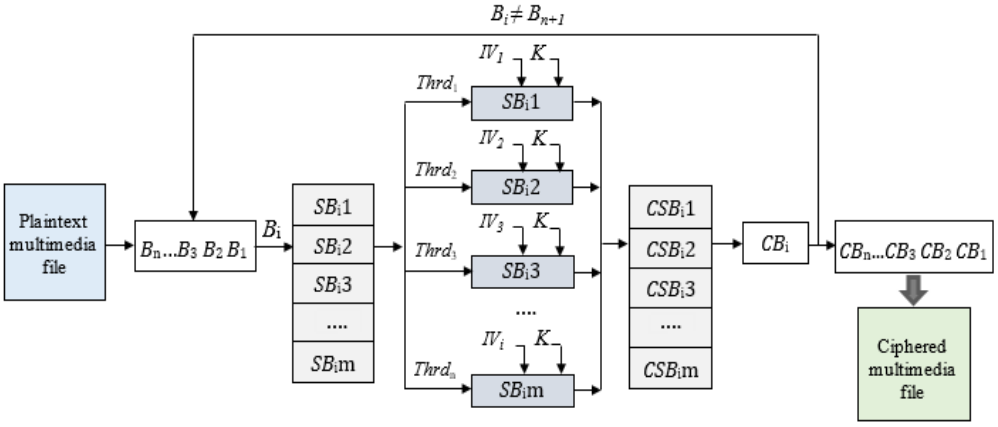


Figure 5. Parallel encryption processing on ParallelFS.

Algorithm 3: Parallel multimedia file decryption

Input: *CK, CF, Prk*
Output: *F*
Begin
//Symmetric key decryption using RSA-2048
K, FSalt ← *Decrypt* (*RSA* (*CK*))
//Parallel sub-blocks decryption using CPU cores
for $\forall CB_i \in F$ **do**
for $\forall CSB_i \in CB_i$ **do**
$ctr_i \leftarrow Fetch$ ( )
$IV_i \leftarrow FSalt \oplus ctr_i$
Connect CSB_i *to* $Thrd_x$
end for
$B_i \leftarrow ParallelDecrypt$ (*Blowfish* (CSB_i, *K*, IV_i))
end for
$F := (B_1, \ldots, B_n)$
Return *F*
End

5. Performance Analysis

This section discusses the evaluation of the performance of ParallelFS in terms of multimedia file write and read operations. The aim is to evaluate the effect of parallel processing on cryptographic performance and assess the complexity of ParallelFS required to improve the response time. In the experiments, we first evaluated the performance of ParallelFS in writing and reading multimedia files with cryptographic operations using different file sizes. Then, the execution times were compared to normal write and read operations for the same file sizes on the standard Ext4 Linux file system. Next, the performance of ParallelFS was compared with that of cryptographic file systems that perform sequential cryptographic processing. We ran all experiments on a multi-core machine equipped with an Intel Core i5-2450M, 2.5 GHz CPU with two cores inside and two threads per core, 4 GB of main memory, 3 MB of cache, and 320 GB of hard disk at 7200 rpm. The machine had Linux Ubuntu 18.04 64-bit installed, and its file system was Ext4, with a 4 KB block size.

The experiments were conducted on a group of multimedia files, including image, audio, and video files ranging in size from 5 to 50 MB. The value presented in each test result was the average of 20 repeated runs, and the file cache was flushed after each experiment to ensure the accuracy of the results. The execution times of different operations were measured and recorded using Python cProfiler [35], which was also used to analyze the statistics to determine and address bottlenecks within the ParallelFS code.

First, we measured the multimedia file read and write time performance in parallel using fork- and thread-based parallelism methods. In the fork-based method, a fork of

multiple processes was used, and in the thread-based method, a single process of multiple threads was used. In both methods, a pool of four processes/threads was created. Moreover, in all parallel experiments, a block of 4 KB was segmented into four sub-blocks of 1 KB each and processed concurrently using Blowfish-CTR.

To measure the read and write times, we used an image file of 8 MB and created multiple process pools using the first method. The first process pool had a single process of four threads; each thread processed 1 KB of data segments. However, these threads were independent of each other's control within the process. The second process pool had two processes and two threads each, whereas the third process pool has four processes with a single thread each. In the second method, we created a pool of a single process of four dependent threads sharing the same resources through global variables. Figure 6 compares the times elapsed for writing and reading an image file in ParallelFS using the two processing method scenarios. The second method of a single process and multiple dependent threads exhibited the best performance of approximately 14.8% for file read operations and 11% for write operations. This was significant because of the intercommunication overhead between multiple processes and threads, the creation of virtual memory, and the associated management overhead. Thus, the context switches between threads in the second method seem to be cheaper than those that occur between the multiple processes in the first method. Consequently, the parallelism in ParallelFS was implemented based on the threading approach of the second method.

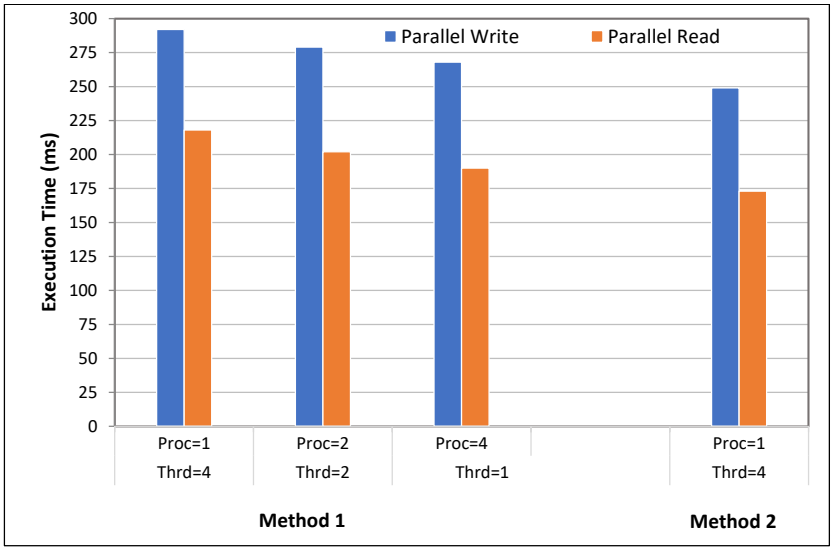
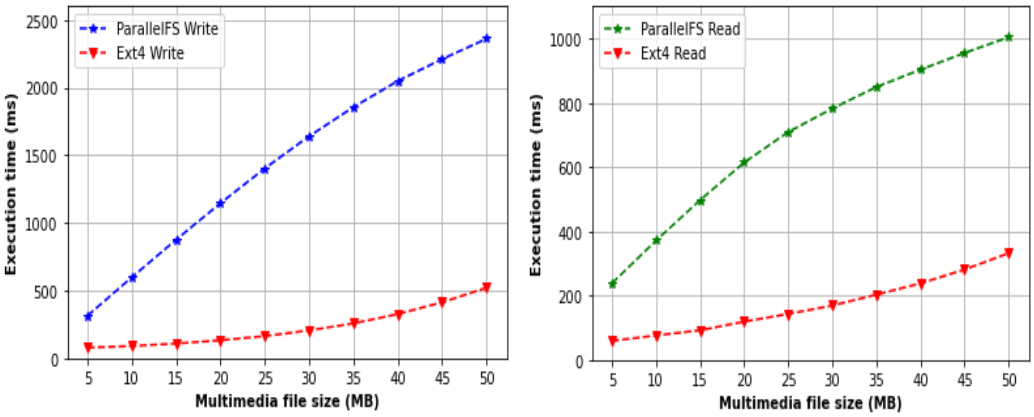


Figure 6. Comparison of parallel writing and reading of an 8 MB file using different numbers of processes and threads with Methods 1 and 2.

Next, we compared the performance of ParallelFS over multimedia file writes and reads with cryptographic services with normal write and read processes without encryption using the standard Ext4. We began the experiment by writing and reading a file of 5 MB and repeated the tests regularly by increasing the file size to 50 MB. In addition, we flushed the file cache for each test. Accordingly, the elapsed times for all the read and write operations were recorded. Figure 7 compares the total times measured for writing and reading multimedia files using ParallelFS and Ext4. ParallelFS can achieve average throughputs of 19.2 and 37.1 MB/s for writing and reading files, respectively, with cryptographic protections. By contrast, the average throughputs for the normal processes to write and read same-sized files using the standard Ext4 were 123.2 and 155.1 MB/s, respectively. Moreover, we calculated the execution time that elapsed during the multimedia file read and write operations in ParallelFS. These times include the time of the actual encryption and decryption operations and that of the other required file system processes performed inside the kernel and at the user level. The related write/read times of other executed processes

include the time spent seeking the writing/reading blocks of file data, the I/O time spent on writing/reading data blocks into a local buffer to perform the encryption/decryption task, the workload time spent on encrypting or decrypting the symmetric keys, and time spent saving or extracting keys from the file header.



(**a**) (**b**)

Figure 7. Comparison of execution times of (**a**) writing and (**b**) reading multimedia files using ParallelFS versus standard Ext4.

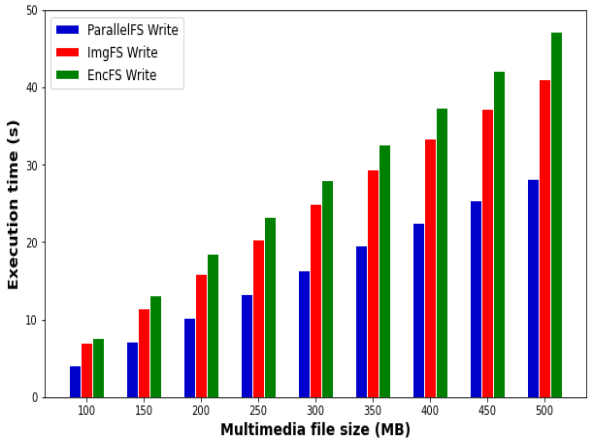
When calculating the time taken by the major executed processes, the actual encryption process accounted for approximately 70% of the real write time on ParallelFS. The other related writing processes on ParallelFS required 30% of the real write time, divided as follows: 21.7% for the I/O write process, 7.4% for the write seek process, and slightly less than 1% for loading public key and saving header processes. Moreover, the actual decryption process on ParallelFS required an average of 75% of the real read time. The other related read processes accounted for 25% of the real read time, divided as follows: 21.6% for the I/O read process, 2.5% for the read seek process, and slightly less than 1% for loading the user's private key and parsing the header.

We tested ParallelFS against benchmarked cryptographic user space file systems using ImgFS [17] and EncFS [24] in writing and reading multimedia files with cryptographic operations, as shown in Figure 8. These results indicate that ParallelFS outperforms the benchmarked file systems for both write and read operations. When calculating the average performance, ParallelFS improves the response time for writing multimedia files with efficiencies of 33.3% and 41.2%, and read efficiencies of 26.4% and 17.6% of the write and read performance using ImgFS and EncFS, respectively.

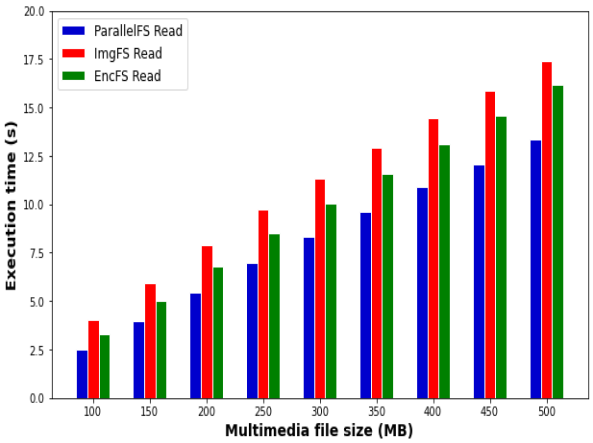
During the development of ParallelFS, we have taken into account that the designed cryptographic file system should provide a higher security level against malicious attacks and meet security requirements. Storage security is a long-term requirement because the attacker has a long time to analyze and break the security system; this is unlike ephemeral transmission, where security is required during the time of data transmission [36]. In ParallelFS, legitimate user authentication is protected. Each user has a unique login passphrase for entering a secure mounting session, and ParallelFS must validate the authenticity of the user before each file system mount.

The confidentiality of multimedia files is ensured, as ParallelFS automatically converts the plaintext into a ciphertext; therefore, it is extremely difficult to recover the original files without proper encryption keys and encryption parameters. In ParallelFS, when a new multimedia file is stored on the disk for the first time, the file is encrypted with a unique symmetric key, which is then encrypted by the asymmetric encryption. Moreover, the malicious user is still unable to recover the files using a brute-force attack, as they need to know the random values of the file's salt and the IVs associated with the file blocks. Data freshness is indispensable for data storage encryption in reducing various types of attacks. In ParallelFS, each block of a file is encrypted differently due to the

unique IV used with each block, thus preventing file blocks from being encrypted to the same ciphertext blocks each time the file is encrypted. Furthermore, the attacker cannot recover the encryption parameters of a multimedia file from other stored files using an offline dictionary attack because new parameters are used with each encrypted file. Key generation and management are transparently managed and controlled by ParallelFS. Secret keys encryption parameters are asymmetrically protected with the public key of the files' owner; therefore, the user cannot determine the secret keys related to multimedia files that he is not authorized to read. We conclude that the proposed ParallelFS is highly secure and can effectively resist attacks and meet the desired security requirements. Table 1 compares the security and efficiency features of ParallelFS with related FUSE-based cryptographic file systems.



(**a**)



(**b**)

Figure 8. Comparison of execution times of (**a**) writing and (**b**) reading multimedia files using ParallelFS versus benchmarked user space file systems.

Table 1. Comparison of security and efficiency features of ParallelFS with FUSE-based approaches.

Feature	CFS [23]	ImgFS [17]	EncFS [24]	SafeFS [26]	OutFS [13]	Proposed ParallelFS
Cryptography	DES cipher using ECB and OFB modes	Hybrid Blowfish and RSA ciphers using OFB mode	AES and Blowfish symmetric ciphers	AES cipher using CBC mode	Hybrid AES and BF-IBE ciphers	Hybrid Blowfish and RSA ciphers using CTR mode
Encryption key length	56-bit	Blowfish 128-bit & RSA 1024-bit	192-bit	AES 128-bit	AES 128-bit & BF-IBE 160-bit	Blowfish 128-bit & RSA 2048-bit

Table 1. *Cont.*

Feature	CFS [23]	ImgFS [17]	EncFS [24]	SafeFS [26]	OutFS [13]	Proposed ParallelFS
Fine-grain cryptographic level	Entirely mounted directory files	Individual image file	Entirely mounted directory files	Entirely mounted driver	Individual file	Individual multimedia file
Cryptographic processing style	Sequential	Sequential	Sequential	Sequential	Sequential	Parallel
Transparency level	Partially transparent	Fully transparent	Partially transparent	Partially transparent	Fully transparent	Fully transparent
Multi-user system	No	Yes	No	No	Yes	Yes
Data freshness	No	Yes	Yes	No	Yes	Yes
Key refreshment	No	Yes	No	No	Yes	Yes
Attack resistance	Moderate	Strong	Moderate	Moderate	Strong	Strong
Efficiency	Low efficiency	Moderately efficient	Moderately efficient	Low efficiency	Moderately efficient	Highly efficient
User convenience	Slightly convenient	Highly convenient	Moderately convenient	Slightly expensive	Highly convenient	Highly convenient

6. Conclusions and Future Work

This study presented the development of a parallel cryptographic user space file system called ParallelFS to reduce the cryptographic overheads incurred during the reading and writing of large multimedia files by leveraging the parallelism of multi-core processors. ParallelFS was designed to perform cryptographic and key management operations in a fully dynamic manner that is completely transparent to users. Parallelization was performed by dividing each 4 KB file block into four sub-blocks. Each sub-block was processed independently, and all were encrypted or decrypted concurrently. We used two methods to execute parallelism: the forking approach, which involves several independent sub-processes through an IPC using virtual memory, and the threads approach, with several dependent threads using global variables. However, we determined that the latter approach had an efficiency that was approximately 11% higher than that of the former approach. The performance of ParallelFS was demonstrated on multimedia files' read and write operations and then compared with related schemes. We compared the performance of ParallelFS against related user space encryption file systems, and the results proved that ParallelFS is able to outperform them. It can effectively improve the writing performance of multimedia files with parallel cryptographic protection, with an efficiency up to 35%, and the reading performance by about 22%, as compared to normal sequential encryption processing in FUSE-based benchmarks. Our experiments indicated that ParallelFS achieved the goal of this study: to attain a higher processing speed with a reduced response time.

Several performance overhead factors account for much of the ParallelFS runtime, namely, task creation, the distribution among processes and threads, termination, interprocess communication, the splitting and collecting of fragments of file blocks, and the limited cache size of the FUSE library. In future work, we plan to refine the performance of ParallelFS by improving on these factors and using more intelligent techniques.

Author Contributions: Methodology, O.A.K. and N.M.K.; Software, O.A.K. and N.M.K.; Validation, W.A. and S.A. (Samer Atawneh); Formal analysis, O.A.K. and W.A.; Resources, N.M.K., W.A., M.A., S.A. (Sultan Alamri), S.A. (Samer Atawneh) and M.K.A.; Data curation, M.A.; Writing—original draft, O.A.K. and N.M.K.; Writing—review & editing, O.A.K. and N.M.K.; Visualization, N.M.K., W.A., M.A. and M.K.A.; Supervision, O.A.K. and N.M.K.; Project administration, O.A.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Khashan, O.A.; Khafajah, N.M. Secure Stored Images Using Transparent Crypto Filter Driver. *Int. J. Netw. Secur.* **2018**, *20*, 1053–1060.
2. Khashan, O.A.; Zin, A.M.; Sundararajan, E.A. Performance study of selective encryption in comparison to full encryption for still visual images. *J. Zhejiang Univ. Sci. C* **2014**, *15*, 435–444. [CrossRef]
3. Saračević, M.; Sharma, S.K.; Ahmad, K. A novel block encryption method based on Catalan random walks. *Multimed. Tools Appl.* **2022**, *81*, 36667–36684. [CrossRef]
4. Khashan, O.A.; Zin, A.M. An efficient adaptive of transparent spatial digital image encryption. *Procedia Technol.* **2013**, *11*, 288–297. [CrossRef]
5. Zin, A.M. Transparent Encryption Technique for Trusted Computing. *J. Phys. Conf. Ser.* **2019**, *1339*, 012011.
6. Seong, Y.S.; Cho, C.; Jun, Y.P.; Won, Y. Security Improvement of File System Filter Driver in Windows Embedded OS. *J. Inf. Process. Syst.* **2021**, *17*, 834–850.
7. Cho, C.; Seong, Y.; Won, Y. Mandatory Access Control Method for Windows Embedded OS Security. *Electronics* **2021**, *10*, 2478. [CrossRef]
8. Soriano-Salvador, E.; Guardiola-Múzquiz, G. SealFS: Storage-based tamper-evident logging. *Comput. Secur.* **2021**, *108*, 102325. [CrossRef]
9. Guardiola-Múzquiz, G.; Soriano-Salvador, E. SealFSv2: Combining storage-based and ratcheting for tamper-evident logging. *Int. J. Inf. Secur.* **2022**, 1–20. [CrossRef]
10. Franzen, F.; Andreas, M.; Huber, M. FridgeLock: Preventing Data Theft on Suspended Linux with Usable Memory Encryption. In Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy, New Orleans, LA, USA, 16–18 March 2020; pp. 215–219.
11. Zhang, Y.; Duan, S.; Zhang, D.; Ren, J. Transparent computing: Development and current status. *Chin. J. Electron.* **2020**, *29*, 793–811. [CrossRef]
12. Bhatt, G.; Bhavsar, M. Performance consequence of user space file systems due to extensive CPU sharing in virtual environment. *Clust. Comput.* **2020**, *23*, 3119–3137. [CrossRef]
13. Khashan, O.A. Secure outsourcing and sharing of cloud data using a user-side encrypted file system. *IEEE Access* **2020**, *8*, 210855–210867. [CrossRef]
14. Khafajah, N.M.; Seman, K.; Khashan, O.A. Enhancing the adaptivity of encryption for storage electronic documents. *Int. J. Tech. Res. Appl.* **2014**, *2*, 28–32.
15. Khashan, O.A.; Zin, A.M. *Transparent Cryptography for Storage Images*; UKM Press, Universiti Kebangsaan Malaysia: Selangor, Malaysia, 2020; pp. 8–20.
16. Vangoor, B.K.R.; Tarasov, V.; Zadok, E. To FUSE or Not to FUSE: Performance of User-Space File Systems. *FAST* **2017**, *17*, 59–72.
17. Khashan, O.A.; Zin, A.M.; Sundararajan, E.A. ImgFS: A transparent cryptography for stored images using a filesystem in userspace. *Front. Inf. Technol. Electron. Eng.* **2015**, *16*, 28–42. [CrossRef]
18. Zou, Y.; Chen, C.; Deng, T.; Zhang, J.; Zhu, X.; Chen, S.; Yin, S. User-level parallel file system: Case studies and performance optimizations. *Concurr. Comput. Pract. Exp.* **2022**, *34*, e6905. [CrossRef]
19. Lee, S.; Jho, N.S.; Chung, D.; Kang, Y.; Kim, M. Rcryptect: Real-time detection of cryptographic function in the user-space filesystem. *Comput. Secur.* **2022**, *112*, 102512. [CrossRef]
20. Bijlani, A.; Ramachandran, U. Extension framework for file systems in user space. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19), Renton, WA, USA, 10–12 July 2019; pp. 121–134.
21. Demir, L.; Thiery, M.; Roca, V.; Tenkes, J.M.; Roch, J.L. Optimizing dm-crypt for XTS-AES: Getting the Best of Atmel Cryptographic Co-Processors (long version). In Proceedings of the SECRYPT 2020-17th International Conference on Security and Cryptography, Lieusant, Paris, 8–10 July 2020; pp. 1–11.
22. Brož, M.; Patočka, M.; Matyáš, V. Practical cryptographic data integrity protection with full disk encryption. In Proceedings of the IFIP International Conference on ICT Systems Security and Privacy Protection, Poznan, Poland, 18–20 September 2018; pp. 79–93.
23. Blaze, M. A cryptographic file system for Unix. In Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS'93), Fairfax, VA, USA, 3–5 November 1993; pp. 9–16.
24. Gough, V. EncFS. 2004. Available online: https://github.com/vgough/encfs (accessed on 10 December 2022).
25. Leibenger, D.; Fortmann, J.; Sorge, C. Encfs goes multi-user: Adding access control to an encrypted file system. In Proceedings of the 2016 IEEE Conference on Communications and Network Security (CNS), Philadelphia, PA, USA, 17–19 October 2016; pp. 525–533.
26. Pontes, R.; Burihabwa, D.; Maia, F.; Paulo, J.; Schiavoni, V.; Felber, P.; Mercier, H.; Oliveira, R. Safefs: A modular architecture for secure user-space file systems: One fuse to rule them all. In Proceedings of the 10th ACM International Systems and Storage Conference, Haifa, Israel, 22–24 May 2017; pp. 1–12.
27. Yoshimura, T.; Chiba, T.; Horii, H. EvFS: User-level, Event-Driven File System for Non-Volatile Memory. In Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19), Renton, WA, USA, 8–9 July 2019.
28. Vangoor, B.K.R.; Agarwal, P.; Mathew, M.; Ramachandran, A.; Sivaraman, S.; Tarasov, V.; Zadok, E. Performance and resource utilization of fuse user-space file systems. *ACM Trans. Storage* **2019**, *15*, 1–49. [CrossRef]

29. He, X.; Long, Y.; Zheng, L. A Transparent File Encryption Scheme Based on FUSE. In Proceedings of the 2016 12th International Conference on Computational Intelligence and Security (CIS), Wuxi, China, 16–19 December 2016; pp. 642–645.
30. Schmuck, F.; Haskin, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In Proceedings of the Conference on File and Storage Technologies (FAST 02), Monterey, CA, USA, 28–30 January 2002.
31. Carns, P.H.; Ligon, W.B., III; Ross, R.B.; Thakur, R. PVFS: A Parallel File System for Linux Clusters. In Proceedings of the 4th Annual Linux Showcase & Conference (ALS 2000), Atlanta, GA, USA, 10–14 October 2000.
32. Khashan, O.A.; Ahmad, R.; Khafajah, N.M. An automated lightweight encryption scheme for secure and energy-efficient communication in wireless sensor networks. *Ad Hoc Netw.* **2021**, *115*, 102448. [CrossRef]
33. Federal Information Processing Standards Publication. 2015, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions; Federal Information Processing Standards Publication. In *Information Technology Laboratory*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2015.
34. Khashan, O.A. Parallel proxy re-encryption workload distribution for efficient big data sharing in cloud computing. In Proceedings of the 2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 27–30 January 2021; pp. 0554–0559.
35. Python Standard Library. The Python Profilers. Available online: http://docs.python.org/2/library/profile.html (accessed on 13 October 2022).
36. Khashan, O.A.; Khafajah, N.M. Efficient Hybrid Centralized and Blockchain-based Authentication Architecture for Heterogeneous IoT Systems. *J. King Saud Univ.-Comput. Inf. Sci.* **2023**, *35*, 726–739. [CrossRef]