Speculative Encryption on GPU Applied to Cryptographic File Systems

Vandeir Eduardo, *Federal University of Paraná and University of Blumenau*;
Luis C. Erpen de Bona and Wagner M. Nunan Zola, *Federal University of Paraná*

https://www.usenix.org/conference/fast19/presentation/eduardo

This paper is included in the Proceedings of the
17th USENIX Conference on File and Storage Technologies (FAST '19).

February 25–28, 2019 • Boston, MA, USA

978-1-939133-09-0

Speculative Encryption on GPU Applied to Cryptographic File Systems

Vandeir Eduardo[1,2], Luis C. Erpen de Bona[1], and Wagner M. Nunan Zola[1]

[1]Federal University of Paraná
[2]University of Blumenau

Abstract

Due to the processing of cryptographic functions, Cryptographic File Systems (CFSs) may require significant processing capacity. Parallel processing techniques on CPUs or GPUs can be used to meet this demand. The CTR mode has two particularly useful features: the ability to be fully parallelizable and to perform the initial step of the encryption process ahead of time, generating encryption masks. This work presents an innovative approach in which the CTR mode is applied in the context of CFSs seeking to exploit these characteristics, including the anticipated production of the cipher masks (speculative encryption) in GPUs. Techniques that demonstrate how to deal with the issue of the generation, storage and management of nonces are presented, an essential component to the operation of the CTR mode in the context of CFSs. Related to GPU processing, our methods work to perform the handling of the encryption contexts and control the production of the masks, aiming to produce them with the adequate anticipation and overcome the extra latency due to encryption tasks. The techniques were applied in the implementation of EncFS++, a user space CFS. Performance analyzes showed that it was possible to achieve significant gains in throughput and CPU efficiency in several scenarios. They also demonstrated that GPU processing can be efficiently applied to CFS encryption workload even when working by encrypting small amounts of data (4 KiB), and in scenarios where higher speed/lower latency storage devices are used, such as SSDs or memory.

1 Introduction

In the era of storing data in cloud services, where they end up being written to server disks scattered around the world, it is increasingly important to deal with data confidentiality before it is actually stored. Cloud storage services typically provide secure communication channels in the data transfer process. However, they do not bother encrypting the files before writing and transferring them, which usually results in data being stored in clear text format.

One way to get around the problem of storing files in clear text is to use a Cryptographic File System (CFS). By using cryptographic techniques, they act transparently, encrypting the data before they are actually stored. CFSs can apply encryption functions at different levels of the data storage and retrieval process, and can encrypt individual files, directories, partitions, and entire disks. CFSs typically encrypt the contents of files in blocks, in order to allow accesses at random without having to decipher them completely. They use block ciphers and modes of operation that dictate specific rules on how the encryption process should be performed.

CFSs have intensive processing demands due to the volume of data and the cost of cryptographic functions. Parallel processing techniques on CPUs or GPUs can be used to meet these demands. As a way to achieve better performance on these systems, you can use the parallel processing capabilities offered by multiprocessor computers and servers, whether in the form of multiple CPUs or GPUs.

The acceleration of symmetric GPU encryption algorithms is a well-studied subject, including the Advanced Encryption Standard (AES) [1] [19] [18] [13] [6] [12], in addition to its application in the context of CFSs [26] [14] [10] [25]. One of the studies related to the acceleration of AES in GPUs resulted in the AES (WAES) *Warped*, presenting significant performance gains [28]. This study also resulted in the creation of the WAESlib library, which can be used to facilitate the integration of applications with the use of AES cryptographic processing in GPUs.

A major feature of WAES, apart from GPU processing, is the use of the operating mode called *Counter* (CTR). The CTR mode has two particularly useful features: the ability to be fully parallelizable in both encryption and decryption operations and the ability to perform the initial step of the encryption process in advance, generating what we call the *encryption masks*.

In addition to exploring the first characteristic, WAES also explores the second, making it capable of computing encrypted data in advance. So when an application actually needs encrypted data, it will already be available, ready to

be used. This feature may prove useful in a number of situations, allowing effective GPU encryption of small amounts of data, something indicated to be impracticable in previous research [26] [14] [10] [25]. Since most file system have blocking factor of 4 KiB or lower, using WAES can bring interesting results.

However, endowing a CFS with the ability to process its cryptographic functions by a parallel processor is not restricted to the simple use of a library. For this to be done efficiently, there are significant challenges to overcome.

A first concern would be about how to implement the CTR mode in a CFS respecting the security requirements required by the mode. It is necessary to treat issues related to an essential element to the operation of the CTR mode called *nonce*. These issues pertain to their generation, storage and management. The resulting implementation needs to ensure that the use of the CTR mode does not cause negative impacts to the CFS performance, which could nullify the gains from parallel processing the cryptographic functions.

Another point with direct connection to the parallel processing of the encryption workload regards to how to manage the CFSs encryption contexts, controlling the production of encryption masks on the multicores or GPU for subsequent use in data encryption and decryption tasks. The read and write operations, either sequential or random, have different characteristics, which requires the creation of different techniques for managing these contexts in order to produce the encryption masks with adequate time in advance. A related issue is about how to aggregate enough work for the cores, for parallel processing from typically sequential workloads generated by each CFS client application, without compromising latency on the CFS operations.

This paper presents techniques that try to overcome these challenges and constitute the main contribution of this work. The authors are unaware of previous work that has used the CTR mode in the context of CFSs, mainly seeking to exploit the advantage of being able to produce the encryption masks in a speculative way. The main objective of such an approach is to obtain a better performance of the CFS in order to achieve higher throughput and more efficient use of CPUs.We specifically deal with aspects related to the counter mode encryption. Authentication issues could be dealt with at FS level or also in conjunction to the encryption processes. In the latter case, our techniques could also be applied in conjunction to other authenticated encryption methods (e.g. GCM [3] or OCB [22]) that are parallelizable and work with similar counter mode constraints.

The techniques presented were validated through the creation of the EncFS++ CFS, based on the preexisting CFS called EncFS [7]. The results show that it was possible to obtain significant performance improvements (in throughput and CPU efficiency) in several scenarios, even in very low latency environments where the base file system (FS) was stored in memory. In SSD disk micro-benchmarks this improvement reached a maximum of 269% in throughput and 112% in CPU efficiency for sequential writing of large blocks. Indeed, in the most adverse scenario for SSDs, the random reading, it was possible to achieve gains of 18% in throughput and 18% in CPU efficiency. The results also demonstrate that the applied techniques allowed performance improvements even when processing small amounts of data (requests of 4 KiB). Our approach was also evaluated in a scenario of low latency "devices", by locating the tested file system in memory (RAM). We also present synthetic macro-benchmarks performed in solid-state disks.

The rest of this paper is organized as follows: In Section 2 we introduce background aspects related to CTR encryption mode, cryptographic storage systems and the GPU encryption library used in our work. We discuss related work in Section 3. Issues related to CTR implementation and storage on CFSs are shown in Section 4. The management of encryption contexts is presented in Section 5. We show micro and macro-benchmark evaluations of our proposed scheme in user space CFS EncFS++ in section 6. Finally we present our conclusions in Section 7.

2 Background

This section presents some concepts relevant to the understanding of this work. The operation of the CBC and CTR modes is discussed, emphasizing the advantages of the CTR mode. Examples of cryptographic storage systems and the different levels at which they can act on the Linux IO subsystem are also presented. EncFS is also previously presented considering that the techniques of this work are implemented on this. Finally, we present WAES, and its WAESlib library, which exploit the advantages of CTR mode and allow the execution of cryptographic functions in GPUs.

2.1 CBC and CTR Operation Modes

CFSs encrypt files in fixed-size blocks so that you can access parts of them without having to encrypt or decrypt them completely. The blocks are coded using block ciphers that subdivide these blocks into smaller blocks (usually 64 or 128 bits), processing them individually [16] [20] [23]. The processing of these smaller blocks during encryption must follow specific rules which are known as operation modes. Among these modes are the CBC and CTR [2].

In CBC mode, each cipher block depends on the clear text block (P), the key (K), and the previous ciphertext block (C). In the encryption process, before a block is encrypted, it undergoes an operation of `XOR` with the previously encrypted block. The encryption operation can be seen in Figure 1a. The CTR mode employs a counter that is incremented for each new block processed. This counter, called *nonce*, is encrypted and then used in a `XOR` operation with the clear text to produce the ciphertext, as in Figure 1b.

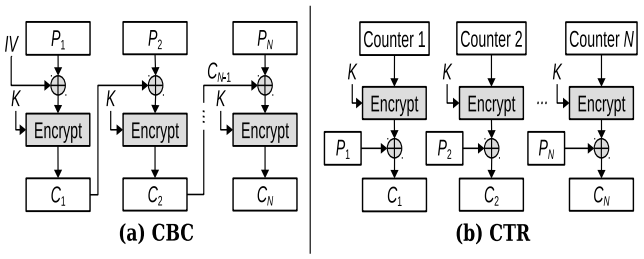

Figure 1: CBC and CTR encryption process.

The CBC mode requires the use of an initialization (IV) vector. Data in IV should not be predictable, ie an attacker with use of some clear text should not be able to predict the IV that will be used in the future process of encryption. The choice of data to be used to populate the IV should follow specific rules. Generally, a pseudorandomly generated number is used, which is encrypted with the same key used in the remainder of the encryption process [20] [23] [4] [2].

In CTR mode, the use of *nonce* enforces a unique nonce and key pair. That is, the same *nonce* can not be used in different encryption operations with a given key in the encryption process. This leads to the explicit need to control nonce increment between different encryption processes that use the same key [2].

With regard to the parallel execution capability of the blocks, in CBC this is not possible. This is due to the fact that the encryption of a particular block depends on the encryption of the previous block. Only the decryption process can be parallelized. In contrast, the CTR mode is highly parallelizable since there is no dependence between the blocks. In addition, since encryption can be done only on *nonce* it is possible to generate encryption masks in advance. Thus, these masks will be ready to be used when the data is known.

Besides the efficiency characteristics of CTR, it has proven security bounds. In fact, the concrete security bounds one gets for CTR-mode encryption, using a block cipher, are no worse than what one gets for CBC encryption [15].

2.2 Cryptographic Storage Systems

Software-based storage systems can operate at different levels of the Linux I/O subsystem. For this reason, the integration of cryptographic resources into this system can be done in several ways, giving rise to different types of cryptographic storage systems.

There are file systems that act on user space, communicating with kernel space through libfuse and the FUSE module [27]. Because they run in user space, they offer greater flexibility, not requiring elevated user privileges to be configured, used, and even developed and tested. On the other hand, by constantly calling from the user to kernel space, they cause a lot of context changes, which compromises their performance. An example of a CFS of this category is EncFS [7].

Other systems are implemented as kernel modules, communicating directly with the Virtual File System (VFS). By running entirely in kernel space, they significantly reduce the need for context switches, improving their performance. An example of this category is eCryptfs [8].

Another approach is to act on the block layer through the device mapper. In this way the encryption will occur only in an agnostic form acting on blocks to the file system itself. They present the best performance, at the cost of not being as flexible as the systems described above. Dm-crypt exemplifies such a system.

The positioning in the Linux IO subsystem of each of these three system examples can be seen in the Figure 2.

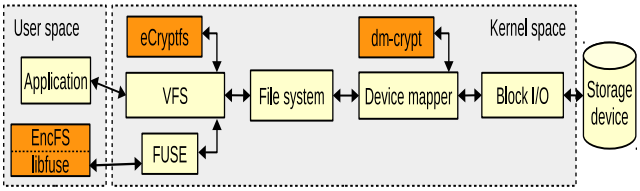

Figure 2: Cryptographic storage systems placement on Linux I/O subsystem.

2.3 EncFS

The implementations of this work were realized on a cryptographic FS called EncFS [7]. The main reason for choosing EncFS was that it is well known and run in user space, which has two important advantages: it facilitates development / testing and allows direct access to the CUDA [17] library, which is only available in user space.

In its encryption processes EncFS can use AES, Blowfish and Camellia symmetric block ciphers available in the OpenSSL library. In order to be able to randomly access data within an encrypted file, its contents are encrypted in blocks with fixed sizes. The size is set during FS creation and can range from 1 to 4 KiB. In full block encryption, it uses CBC mode of operation.

Three different types of IVs are used in block encryption. The first is the IV contained in the volume key (IVV). The second is the IV of the file (IVA) and the third is the IVs for encryption of the blocks within the files (IVB). The IVV is stored in the final bytes of the volume key (VK), which is generated randomly in the FS creation. IVA is also generated randomly in creating a new file. $NumBlock$ corresponds to the block number within the file. IVB is obtained by applying the following cryptographic hash function:

$$IVB = HMAC\_CTX(VK, IVV \parallel (NumBlock \oplus IVA))$$

Because EncFS uses CBC mode, the IVs used in block encryption (IVBs) must meet the unpredictability requirement. This is why the concatenation of different IVs and the

application of the cryptographic hash function is done. Considering the same file stored in a given EncFS system, VK, IVV and IVA remain constant, the only variable being the block number. This feature has advantages over performance, as it allows the IVBs to be dynamically calculated, and it is not necessary to store them. In addition, the same IVB can be reused in rewriting processes of the same block within a same file.

2.4 WAES and WAESlib

WAES (*Warped AES*) [28] is the implementation of a high performance algorithm for heterogeneous parallel processing employed for data encryption using GPU. WAES was implemented with 128-bit and 256-bit keys in CTR mode. The heterogeneous part of WAES refers to the possibility of performing all the steps of the CTR operation mode on the GPU or performing the final part of `XOR` with the text to be encrypted or decrypted in the CPU.

WAES explores the characteristic of the CTR mode of operation to implement a technique called speculative encryption. From the moment you have the encryption key and *nonce*, WAES can compute and pre-populate *buffers* with encrypted data. In this way, these data are available in advance and are called encryption masks.

Anticipating the production of encryption masks in conjunction with performing the final `XOR` step of CTR mode on the CPU are critical to efficient processing when working with small amounts of data in GPUs. This anticipation allows you to compensate for the latency involved in GPU operations such as data transfer and GPU kernel activation. Performing the final `XOR` step on the CPU avoids the need to transfer the data to be GPU encrypted.

WAES also allows buffer aggregation, which are processed in the same WAES kernel activation. The buffers aggregation technique decreases the latency and increases the utilization of the GPU processing cores, consequently resulting in higher bandwidth, thus being a promising technique to be used with file systems submitted to workloads composed of requests in the 4 KiB range.

The techniques proposed by WAES were implemented and made available in a library called WAESlib. Calling this library will prepare the CUDA environment and launch the WAES *kernel* which will pre-compute the encryption masks. Calls to masks made through WAESlib are asynchronous, freeing the application to perform other tasks while the encryption masks are computed on the GPU.

WAESlib also offers a priority feature that can be used to control the production order of encryption masks. Contexts defined with higher priority have their masks produced before. This feature can be explored by trying to sort the production of the masks according to the order in which blocks of a file are accessed.

3 Related Work

The GPU acceleration of symmetric block ciphers is a well-known subject in the literature [1] [19] [18] [13] [6] [12]. Some of these researches aim to take advantage of the acceleration of these ciphers in cryptographic storage systems. In this section we first present research that worked with systems running in user space and in the sequence those that worked with systems running in kernel space.

In user space. The Engine-CUDA [21] project features an OpenSSL engine capable of performing symmetric GPU encryption operations. The project is used in the development of the work done in [6], where the engine has been improved and extended, being implemented other ciphers such as DES, Blowfish, IDEA, Camellia and CAST5. Results show that GPU processing becomes effective above 16 KiB and can be eight times better when approaching 8 MiB.

In [5], CrystalGPU is presented, which is a framework that aims to facilitate the integration of GPU processing into applications. This framework is used in the implementation of a FUSE-based CFS called CRSFS [26] [14]. In this work the encryption algorithms used are the AES operating in the ECB and CBC modes. Experimental results show that it is advantageous to use CPU processing for data sizes up to 4 KiB and GPUs for data above 16 KiB.

In kernel space. The OpenBSD Cryptographic Framework (OCF) [11] is a framework developed with the goal of providing a service virtualization layer within the kernel by offering an API that hides the specific details of each accelerator. The work presented in [9] uses the Linux version of the OCF to integrate the GPU processing resources into this *framework*, allowing to execute in NVIDIA GPUs symmetric encryption operations using AES in ECB mode.

Gdev [10] is a runtime system designed to run in the kernel space to manage the use of the GPU in a way that is similar to the processes running on the CPU. Its main feature is to control the GPU without relying on proprietary drivers and access libraries that run in user space. One of the practical applications demonstrated is the eCryptfs adaptation to perform encryption on the GPU. For writing operations the gain was up to 2x.

The GPUStore [24] [25] is a runtime and framework system designed to facilitate the integration and efficient use of GPU processing for data storage systems that run in *kernel* space. In [25] as a practical application, the GPUStore is used to accelerate code inside the kernel from the dm-crypt, and eCryptfs. For data encryption above 256 KiB, the performance was 36 times better.

Two points can be highlighted in relation to the state of the art presented. First, when applying the processing acceleration of the cryptographic functions in GPU to file systems none of the work took advantage of the CTR mode. To our knowledge, the work proposed in this paper is the first to explore the CTR mode in this context. The second is that results in the reviewed

work indicate that GPU processing only begins to be efficient when working with larger size requests (> 16 KiB).

The use of the CTR mode for CFS proposed here allows new forms of performance gain not exploited in previous work: exploring the ability to produce the encryption masks speculatively in the GPU; and to avoid transferring the data to be encrypted or decrypted from the CPU to GPU. These techniques allow to compensate for the additional latency inherent to the GPU processing that are especially impacting when the size of the requests is small, according to the research data presented. Moreover, we show that the data space needed for nonce storage is very small and that it can be retrieved and operated without compromising performance. Our approach is applicable, with some adaptation, to other *counter mode* encryption methods that include authentication and are parallelizable, including on multicores.

4 CTR Implementation on CFSs

Before we can exploit the benefits of using CTR in the context of CFSs, it must be adapted to operate in CTR mode. One essential element for the operation of the CTR mode is the nonce. We must carefully think the ways to generate and store these nonces as these factors can impact the performance of the CFS and possibly cancel any impending gains from the application of speculative encryption using GPU processing. This section presents some techniques that address such issues.

4.1 Nonces Generation

Considering the issue of *nonces* generation and seeking to satisfy the demand for *nonce* uniqueness [3] in the CTR mode, this work proposes a deterministic form for its generation. The technique relies on a single global counter that is incremented on each write or rewrite of a data block in the FS. The numbers generated by this counter are used as *nonce* in the encryption and decryption functions. Considering, for example, the use of a 128-bit block cipher, the *nonce* must also have that size. Therefore, a 128-bit counter can be used.

In implementations such as the OpenSSL library, the least significant nonce bits are used as an internal counter incremented every 16 bytes of encrypted data. The amount of bits required for this counter is given by the formula $log_2(x/16)$, where x is the number of bytes to be encrypted with this *nonce*. Since *nonces* must be unique, the amount of blocks that can be written or rewritten is given by 2^{128-x}, where x is the amount of reserved bits.

Figure 3 illustrates the format of the global counter and the least significant bits reserved for the internal counter of the CTR mode.

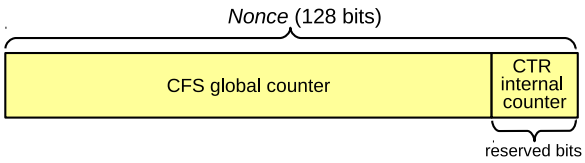

Figure 3: Nonce obtained from CFS global counter.

4.2 Nonces Storage

Regarding the storage of *nonces*, when a FS block is written, the *nonce* used in its encryption process is obtained from the FS global counter. In a future process of reading the same block, in order to be able to decipher its contents, it is necessary that the same *nonce* be passed as a parameter to the decryption functions. Therefore, it must be stored for future retrieval since any rewritten block needs a different *nonce* and file blocks may have been updated in any given order.

A naive method would be to store the *nonces* individually, prepending each block. However, this approach is inappropriate for random reading because it makes it difficult to read *nonces* in advance in order to trigger the anticipated generation of encryption masks. In addition, FS block sizes are already page aligned and doesn't have the space to include the *nonces*.

One way to work around this problem is to store *nonces* separately from the encrypted file. Thus, they occupy contiguous regions on disk, streamlining the processes of reading and writing. Also, since each *nonce* has only 16 bytes, accessing it individually is not efficient. In addition to storing them in separate files, it is also interesting that they be read and written in a clustered fashion.

However, grouped and separate storage is not the ideal solution for all cases. In a scenario that works with very small files, containing few blocks of data, reading and writing an entire group of *nonces* can lead to wasted memory and disk space. Ideally, we would store the initial *nonces* of all files in a single storage location. Only when a file occupies a greater number of blocks, its other *nonces* begin to be stored in a separate and exclusive file.

To deal with the storage situation of the initial file *nonces*, this paper proposes a solution based on the Unix *inodes* storage system. This structure was called *nonce node* (*nnode*). There is a single file responsible for storing the *nnodes* of all files stored in the FS. The general structure of this file can be seen in Figure 4a.

At the beginning of the *nnodes* structure (file) the FS global counter is stored. Following is a counter that controls the amount of *nnodes* used, as well as a bitmap that is intended to control the allocation of *nnodes*. The *nnodes* themselves are in the sequence. Each file stored in the FS has an *nnode* allocated to it and stored in that structure. In the expansion of an *nnode*, shown in Figure 4b, you can see the information contained in it. They are the inode number of the file (used to

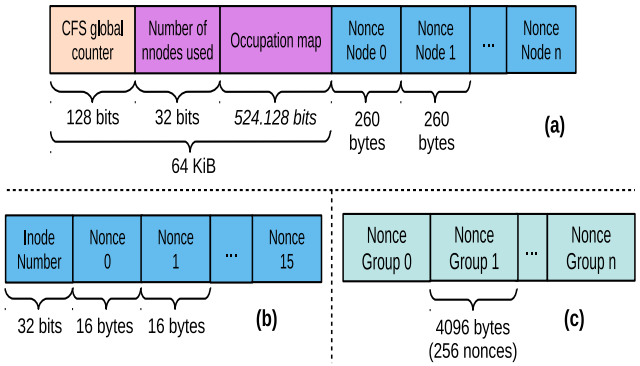

Figure 4: Nonce nodes (nnodes) file format (a). Detail of a nnode (b). Exclusive nonce file format (c).

index the file to *nnode*), followed by the first 16 *nonces* of the file.

Considering that an *nnode* is used to store the first 16 *nonces* of a file, assuming for example that the FS uses 4 KiB blocking factor and supposing files of up 64 KiB, only the structure of one *nnode* is sufficient to store its *nonces*. This form of storage helps to optimize access to small files as it allows nnodes to be stored in near regions on the disk. In addition, it also helps to avoid wasting disk space by allocating larger structures for storing an entire group of *nonces*.

If a file grows beyond the 16 blocks, the other *nonces* begin to be stored in a separate and unique file for each file stored in the FS. The format of this file can be seen in Figure 4c. *Nonces* are stored in groups of 256 *nonces* to match the size of a 4 KiB memory page. This implies that that one *nonce group* is stored for each $256 * 4$ KiB segment of a file (i.e. a 1 MiB file segment). This way the overhead, in terms of disk space to keep the nonces, stays negligibly under 0.4% (i.e. at most $4KiB/1MiB$).

5 Encryption Contexts Management

In order to understand some of the techniques proposed in this work, which aim to use WAESlib efficiently in the context of CFSs, it is necessary to briefly describe its operation. One of the main components of the library is the so called encryption contexts: a kind of logical organization used by the library to control the production and application of encryption masks.

Basically, the use of the library is a process that involves calling a context definition function to start the production of the encryption masks and then call the functions that apply these generated masks.

In the use of the context definition function, the identification number of the context being defined, the identification number of the previously defined key, *nonce* and a priority number are given. The call to this function is asynchronous, releasing the application to perform other tasks. After the function call, the library can fire the WAES *kernel* so that it begins computing the encryption masks in advance in the GPU.

Internally the priority is used by WAESlib to aggregate and define the order in the production of the masks. As the masks are generated, they are transferred to the system memory and are available for use in the data encryption and decryption functions.

The other two main functions are used to instruct WAESlib to encrypt and decrypt data. In your call information such as the context identification number, the *buffer* containing the data to be encrypted or decrypted and its size is passed. These are synchronous functions that cause WAESlib to use the previously calculated encryption mask for the context in question, available in system memory, to effectively encrypt or decrypt the data. This final step consists of a bitwise `XOR` operation between the data and the mask generated, being performed in the CPU.

Despite its simplified use, there are significant challenges involved in this process. They mainly concern how to effectively group, define, and use these encryption contexts in writing and reading operations. One of the main contributions of this paper is to present some techniques that show how to do this. They demonstrate how to use encryption contexts to ensure that encryption masks are produced promptly in advance and are readily available at times when they are required to effectively encrypt and decrypt blocks of data. These techniques seek to explore how files stored in the CFS are accessed, for example by examining data locality issues and access patterns.

5.1 Encryption Context Pools

The approach carried out in this work in grouping the encryption contexts was based on the idea of pools that are used differently according to the operation being performed. We identified the need to create at least two group types: a unique *pool* of encryption contexts, at CFS level, used in sequential and random write operations; and several *pools* of contexts for decryption, one per open file, used in sequential and random read operations. The working idea of these pools is described in the following two subsections.

5.1.1 Contexts Pool for Encryption/Writing

In order to hide the latency involved in the process of generating the masks in the GPU, as well as its transference from the memory of the device to the system memory, it is essential that the triggering of the masks generation occurs as soon as possible. This work proposes the use of a single *pool* of contexts used for the generation of masks that can be used in the process of block encryption. Consequently, they can be used in the processes of writing and rewriting blocks of all CFS files.

Using as an example a *pool* containing eight contexts, when the CFS is mounted, eight contexts will be allocated for this *pool*. The current value of the CFS counter is copied and used to define the contexts contained in the *pool* using a *nonce* corresponding to the value of that counter, which is incremented with each defined new context. Respecting the need to keep the least significant bits of the counter reserved and considering a CFS that uses blocks of 4 KiB, the counter increases in the value of 256 (as each 4 KiB data block totals 256 AES blocks).

The queue start (pool beginning) indicator is used to store the context ID containing the oldest mask generated. At the end of this process, shown in Figure 5a, the CFS will have an amount corresponding to the size of the pool in masks ready to be used in block encryption processes.

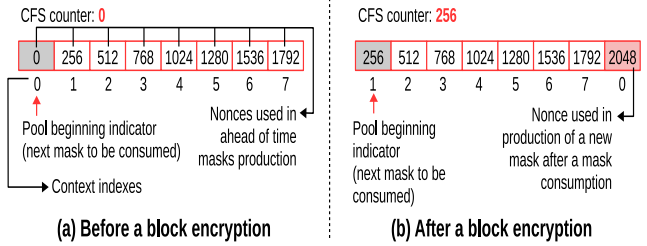

Figure 5: Contexts pool usage example for encrypting blocks in write operations.

However, in order to avoid wasting *nonces* and avoid the need to temporarily store the *nonces* used to generate masks, the CFS counter is only effectively incremented when a mask is used in the encryption process. The *nonce* to be saved to be used in the future process of decrypting the block is obtained from the value of the CFS counter when mask usage occurs. Soon after, the counter is incremented, corresponding to the value used before in the generation of the mask of the subsequent context. The queue start indicator is then moved.

After consuming the mask, the context that contained the newly used mask is reused to trigger the generation of a new mask. Thus, all contexts are always maintained with a new mask. The priority used in this context reset (redefinition) may be as small as possible, since the mask to be produced will only be consumed after all others. The value of *nonce* to be used in the generation of the next mask can be obtained through the formula $x = y + (1 << w) * z$, where x corresponds to the next *nonce* y the current value of the CFS counter, w the amount of reserved bits of CTR mode and z the size of the *pool* of writing contexts. Figure 5b illustrates the state of *pool* after consumption of a mask, followed by generation of a new mask.

5.1.2 Contexts Pool for Decryption/Reading

The *context pool* used for decryption, and consequently in the reading processes, works as a mask window that is shifted according to the region of the file being read. In this technique, each open file has its unique *context pool* whose trigger for masking occurs according to the block number being accessed. The window is positioned on the first read block, generating the masks for the subsequent blocks according to the size of the *pool*. As the blocks are read and the masks consumed in the decryption processes, the window is shifted and new masks are generated.

The Figure 6 illustrates the behavior of the *pool* of contexts containing the generated masks in a sequential read process, initiating at the beginning of the file. Also taking as an example a *pool* containing eight contexts, when the file is opened, the *nonces* of the first eight blocks are used to define the encryption contexts that begin to produce the masks.

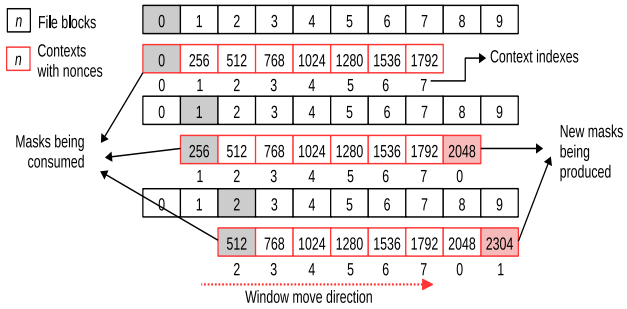

Figure 6: Contexts pool usage example for decrypting blocks in sequential read operations.

The priority feature offered by WAESlib can be exploited to dictate the production order of these masks. Thus, when defining each of the contexts of the window, one can define the context contained at the beginning of the window with the highest possible priority, gradually decreasing this priority in the definition of subsequent contexts. The masks needed to decipher the first blocks are getting ready before the masks of subsequent blocks.

As illustrated in Figure 6, after consumption of the mask referring to the $0th$ block, one can trigger the production of the mask for block eight with the lowest possible priority, since the trend is that it will be used only after consumption of the masks from the previous blocks. In a purely sequential reading, this process repeats itself, with the consumption of the mask at the beginning of the window and the subsequent production of a new mask at its end.

The sliding window technique can also be applied for random access. In this case, its beginning is always moved to the position corresponding to the first block being read. When this displacement occurs, two situations may occur at first: (i) the window is fully shifted to an earlier position, i.e. $(x - y) > z$, where x is the previous initial position of the window, y the new starting position and z the size of the window; (ii) the window is shifted fully forward, where $(y - x) > z$. Both situations are illustrated in Figure 7a and 7b, respectively. Because the new starting positions are completely outside the previous

window, it is necessary to reset (redefine) all contexts for new masks to be generated.

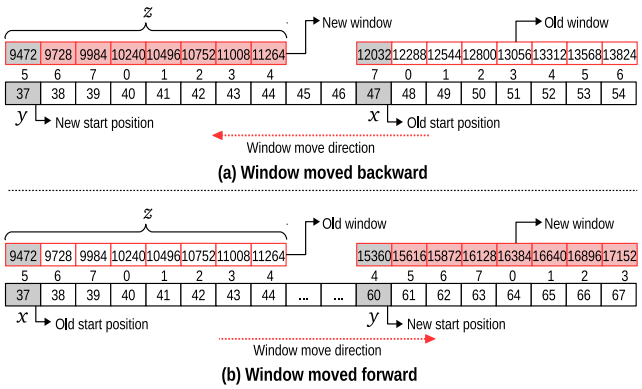

Figure 7: Window moved to a far position with the redefinition of all contexts inside the pool.

The other two situations refer to cases where, after moving the window, part of the new window ends up overlapping the previous one. These situations occur when the window is moved to a near previous position, that is $(x-y) <= z$; and when it is shifted to a near posterior position, that is, $(y-x) <= z$. The figure 8a and 8b illustrates the first and second case, respectively. In these situations, the window that occupied the previous position contains some masks that can be reused, it is not necessary to redefine the contexts of the positions that overlap, making the process more efficient.

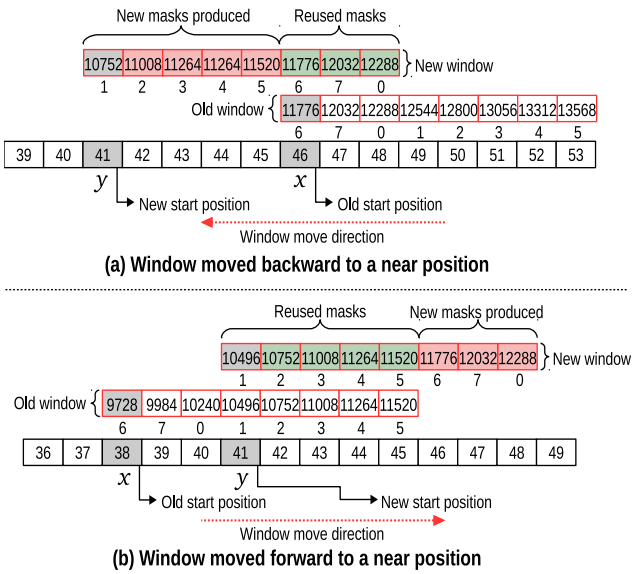

Figure 8: Window moved to a near position with reuse of previously produced masks.

6 Evaluation

As a way of validating the techniques discussed in the previous sections, they were implemented on the EncFS CFS. We have called this new version EncFS++. The performance analyzes in this section compare EncFS and EncFS++. The EncFS version is the original, using CBC mode and with exclusive CPU processing. The EncFS++ version uses the CTR mode implemented according to the ideas discussed in Section 4, as well as the encryption context management techniques for GPU processing discussed in Section 5.

Performance measurements were performed at the micro-benchmark level measuring throughput with 4, 64 and 128 KiB requests in read and write operations both sequentially and randomly. To measure throughput the tool `fio` was used; to measure CPU utilization, the `pidstat` tool was used to exclusively monitor the EncFS and EncFS++ processes and subprocesses. Macro-benchmarks were also performed with the `filebench` tool, as described at the end of this section. The tests were performed on a computer running Linux OS with 4.10 kernel, using Intel Core i7-7700HQ @ 2.8GHz, 32 GiB memory and Western Digital SSD disk model WDS240G1G0A. The libfuse version used was 2.9.4, OpenSSL 1.0.2g and WAESlib 2.01g. The GPU used was an NVIDIA GeForce GTX 1070 (mobile version), in CUDA 9.2 environment. The micro-benchmark measurements were performed on a 16 GiB file, being repeated 10 times and taking the simple arithmetic mean of the results. Among the measurements, the base FS (`ext4`, default settings) was unmounted, re-created and remounted. Among all the repetitions, the caching pages were discarded.

As a way of comparing CPU utilization between versions, an index called CPU utilization efficiency was used. It was calculated using the following formula: $x = (y/z)/(w/k)$, where x corresponds to the efficiency index being calculated, y and z correspond to the throughput and utilization value of CPUs reached in the version being compared; w and k correspond to the value of bandwidth and CPU utilization reached in the version with which the comparison is being made. Thus, it is possible to obtain a comparison between the amount of work performed by each version contrasted with the percentage of CPU usage.

Figure 9 shows the bandwidth values obtained in the two versions of EncFS, in a sequential read scenario with base FS stored on disk (SSD) and in memory. In the secondary y axis (right side) the percentage of the bandwidth variation of EncFS++ is displayed in relation to its original version. Table 1 displays the values obtained. The next to last column shows the percentage increase or decrease in bandwidth. Positive and green values indicate that the EncFS++ version had an increase in bandwidth compared to the original version. Negative values in red means the opposite. The last column shows the calculated efficiency values. Values above 1 were colored green indicating more efficient CPU usage in the

EncFS++ version. Values below 1 were colored red indicating less efficient use.

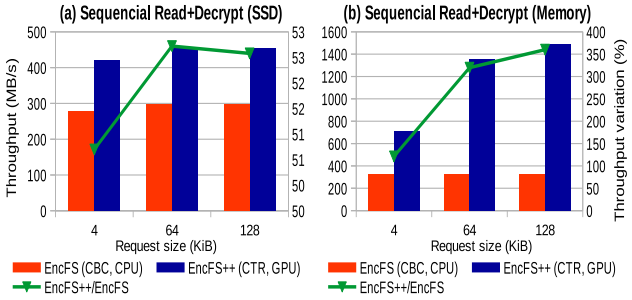

Figure 9: Throughput variation (sequential read+decrypt).

Req. Size (KiB)	Sequencial Read+Decrypt (SSD)						Sequencial Read+Decrypt (Memory)					
	EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/EncFS		EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/EncFS	
	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency
4	278.47	11.32	419.66	14.71	50.70	1.16	323.25	12.20	714.82	21.76	121.14	1.24
64	297.04	12.10	453.64	11.54	52.72	1.60	322.91	12.20	1,355.20	23.90	319.68	2.14
128	297.69	12.11	454.20	11.54	52.58	1.60	323.03	12.20	1,485.59	23.90	359.90	2.35

Table 1: Throughput and CPU utilization: sequential read+decrypt.

With the base FS stored on disk, it is perceived that the maximum throughput gain in EncFS++ is approximately 52%, with an increase in CPU utilization efficiency around 60%. However, the bandwidth number reached approaches the throughput limit supported by the disk (500 MB/s). With base FS stored in memory (Figure 9b), it can be seen that it is possible to obtain much more significant gains, up to 359% in throughput and 135% in CPU use efficiency.

The significant performance improvement in the reading operations deserve to be highlighted, mainly because they involve the use of the decryption functions. CBC mode also has the same advantage as CTR mode in that it can be parallelized in the decryption processes. Therefore, a simple application of the CTR mode, without exploiting the parallel processing in GPU, would not have many advantages with respect to the improvement in throughput.

The results of the sequential writing operations can be seen in Figure 10 and the values in the Table 2. In the sequential writing scenario, EncFS++ also has a significant throughput gain of up to 269%, using base FS on disk, and 324% with FS in memory. CPU utilization efficiency, is up to 212% and 133%, with base FS stored on disk and memory, respectively. In both cases, direct write to disk (`O_DIRECT`) was not used because it was not supported by EncFS. As a consequence, writing occurs first on cache pages. For this reason, the values written to disk and memory are very close and values exceed the bandwidth limit of the SSD disk.

Significant gains in both sequential reading and sequential writing demonstrate the effectiveness of context pool management techniques in order to be able to produce encryption

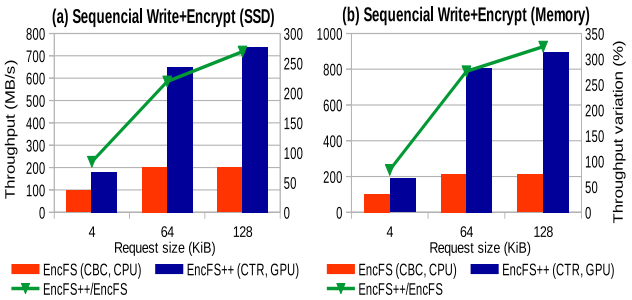

Figure 10: Throughput variation (sequential write+encrypt).

Req. Size (KiB)	Sequencial Write+Encrypt (SSD)						Sequencial Write+Encrypt (Memory)					
	EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/ EncFS		EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/ EncFS	
	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency
4	97.43	10.41	179.86	11.75	84.61	1.64	102.88	10.42	188.11	12.19	82.83	1.56
64	203.79	10.55	650.71	14.16	219.30	2.38	214.71	11.00	806.92	18.59	275.82	2.22
128	199.87	9.93	738.43	11.75	269.44	3.12	211.56	10.43	897.25	18.94	324.12	2.33

Table 2: Throughput and CPU utilization: sequential write+encrypt.

masks well in advance. Even in a scenario of very low latency as in the case of base FS stored in memory. It is also important to highlight the gains obtained in the measurements involving 4 KiB requests, demonstrating that it is also possible to perform GPU processing efficiently when the CFS works with small amounts of data. As described in Section 3, previous research indicates that this is only feasible with larger requests (> 16 KiB).

The results of the random read can be seen in Figure 11 and the values in the Table 3.

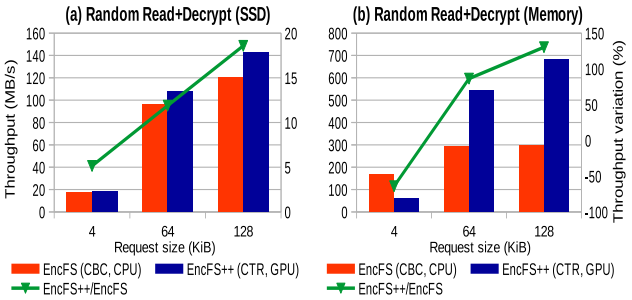

Figure 11: Throughput variation (random read+decrypt).

Req. Size (KiB)	Random Read+Decrypt (SSD)						Random Read+Decrypt (Memory)					
	EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/EncFS		EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/EncFS	
	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency
4	17.80	1.93	18.71	3.75	5.14	0.54	166.68	10.37	59.72	8.30	-64.17	0.45
64	96.48	4.37	107.95	5.35	11.88	0.91	290.74	11.40	541.54	17.12	86.26	1.24
128	120.17	5.34	142.50	5.38	18.57	1.18	297.11	11.45	684.10	17.37	130.25	1.52

Table 3: Throughput and CPU utilization: random read+decrypt.

Before discussing the results of random reading, it is important to explain how the random reading generated by the

`fio` tool occurs. When generating the random accesses, the tool determines the next block to be randomly accessed based on a uniform distribution. While all blocks of a file are not accessed, the reading of a previously held block does not repeat. Since the CFS was configured to encrypt blocks of 4 KiB, requests of 4, 64 and 128 KiB result in access to 1, 16 and 32 blocks, respectively.

The context pool for reading is designed to be used for both sequential and random reading. In sequential reading this pool was configured to contain 64 contexts (window size). That means, in every read request, 64 encryption masks will be produced. In sequential reading, this is not a problem, since the blocks are read in sequence and all masks are used. However, considering the random access of 4 KiB, if a window of 64 contexts is used, mostly the first mask produced is consumed, as the chances of the next block to be read in the sequence is low in random read patterns. Consequently, a significant overhead would have occurred in the production of 63 masks that are not harnessed. Therefore, in random readings, the size of the window was adapted according to the size of the request. For the 4, 64 and 128 KiB requests, windows containing 2, 8 and 16 encryption contexts were used.

First, by analyzing the results with the base FS stored in memory, in the 64 and 128 KiB requests, there are gains in throughput (86% and 130%), but they are smaller than the values obtained in sequential reading (319% and 359%). If you use smaller windows, you also reduce the level of advance that masks are generated when the request is being served. It is also important to highlight the issue of full window offset when a new random read request occurs. All window contexts need to be reset and there is less time for the masks corresponding to the initial blocks of the request to be ready, causing the application to wait for its production.

In the random 4 KiB requests and FS in memory there are significant throughput losses (64%) and low CPU use efficiency. Considering the random access where it is not possible to predict the next block to be accessed and due to the use of a small window (2 contexts), there is no way to shoot the mask production well in advance. Therefore, with each block accessed, the application needs to wait for the mask to be ready, adding latency to the decryption process. The scenario is aggravated by the fact that the base FS is stored in memory, because in this case there is neither the time to read the block on disk, in which, in parallel, the mask could be generated. In Figure 11a, it can be seen that there are no throughput losses in 4 KiB requests, which demonstrates how disk access time can be exploited as a way to compensate for processing latency in GPU kernel launches. One solution to circumvent this outcome is to either lower the latency in GPU decryption library setup or to resort to pure CPU CTR decryption when 4K random read patterns are presented to the FS.

In contrast, according to the performance observed in the 64 and 128 KiB requests, even without having the access time to disk to produce the masks, the use of a slightly larger window (8 and 16 context) and the technique of launching the production of a mask for the block *n* positions ahead (where *n* corresponds to the size of the window), are enough to achieve a certain level of advance in the production of the masks. Thus, even in this scenario of very low latency (without access to disk), we still have gains in throughput.

However, although disk access time can be exploited as a way to compensate for the latency of GPU processing, it also acts against WAESlib's ability to aggregate encryption contexts, mainly in the random read scenario. As a consequence, the efficiency in the use of GPU processing resources is reduced and the overheads inherent in GPU processing (GPU kernel launching and mask transfer) become more significant. Even with these adversities, there are still gains in this case with the base FS stored on disk, although modest, from 5% up to 18% in throughput boost.

Regarding CPU utilization efficiency, in the disk FS scenario, there is a reduction of about 46% (i.e. 0.54 efficiency) in the 4KiB requests, the worst case. In practice, this means that it would not be possible to ameliorate CPU utilization in these small requests sizes, given the throughput levels achieved. We have observed that polling mechanisms, either internal in CUDA or in the GPU encryption library, generate slightly higher CPU overhead when waiting longer for disk operations to complete. This is an issue to be further investigated. One possible solution was dicussed before, for the small 4KiB random read case from memory.

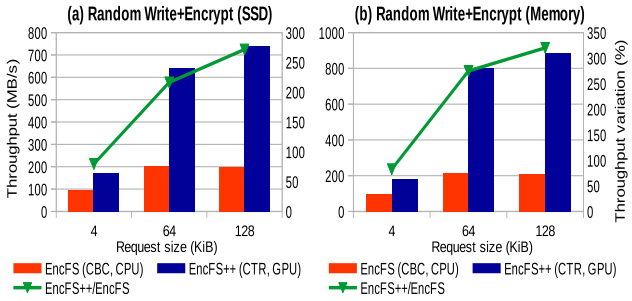

Figure 12: Throughput variation (random write+encrypt).

Req. Size (KiB)	Random Write+Encrypt (SSD)						Random Write+Encrypt (Memory)					
	EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/ EncFS		EncFS (CBC, CPU)		EncFS++ (CTR, GPU)		EncFS++/ EncFS	
	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput Var. (%)	CPU use efficiency
4	96.61	10.36	173.46	11.54	79.54	1.61	99.81	10.28	182.13	11.94	82.47	1.57
64	202.53	10.50	640.83	14.06	216.42	2.36	213.83	10.97	801.79	18.57	274.96	2.21
128	198.91	9.91	739.30	11.60	271.69	3.17	211.47	10.43	888.46	18.89	320.13	2.32

Table 4: Throughput and CPU utilization: random write+encrypt.

The results of the random writing can be seen in Figure 12 and the values in the Table 4. The outcomes in this scenario are very close to the sequential writing scenario. This is because writing occurs in memory regardless of whether the base FS is on disk or memory, as explained before. In

addition, the pool of contexts used in sequential and random writing is the same, including with respect to its mechanism of operation. There are significant gains with both disk (271%) and memory (320%) FS, citing the maximum numbers. There are also improvements in CPU efficiency of up to 217%and 132%, respectively.

On the synthetic macro-benchmark analyses, two example workloads from filebench were used: *fileserver.f* (characterized by a mix of read and writes operations) and *webservef.f* (characterized mainly by read operations). EncFS++ performance was compared to the original EncFS and with eCryptfs. The hardware accelerated (AESNI) as well as the CPU version of eCryptfs were used due to the fact that it is also stacked over *ext*4 FS (figure 2), but operating in kernel space. Measurements with simultaneous access by 1, 2 and 4 threads were performed. In this scenario, the base FS (ext4) was stored on a SSD disk.

Greater throughput gains occurred in the *webserver.f* workload (Figure 13 and Table 5). The gains are up 145% when compared to EncFS and ranges from 44% to 130% with respect to eCryptfs. In the *fileserver.f* workload, the gains were more modest: between 9% and 27% when compared to EncFS and and between 2% and 33% contrasted to eCryptfs. It is important to note that with the base FS stored in SSD, the maximum throughput supported by the disc imposes a limit to the gains. Despite the theoretical limit of 540/465 MB/s (read/write) supported by the SSD, in practice this value is 400/200 MB/s (measured by `dd` tool with `bs=128k`, same size as *iosize* parameter used in the filebench tool). Regarding CPU use efficiency, there were also gains. Compared to EncFS in the *fileserver.f* workload, these gains ranged between 120% and 170% and in *webserver.f* it was between 60% and 80%. The gains against eCryptfs in the *fileserver.f* workload stayed between 110% and 190%, and in *webserver.f* ranged from 90% to 120%.

A comparison with eCryptfs(AESNI), a CFS executing AES encryption in hardware, is a fair experiment with macro-benchmark workload. In particular because the latency at submitting encryption tasks is much lower than spawning GPU work, and this FS works in kernel space. Even in this case, the performance of EncFS++ proved competitive with respect to throughput capacity. EncFS++ showed gains between 8% and 32% in *webserver.f* workload. In the *fileserver.f* workload, the throughput practically stayed the same. The negative point in this scenario was in the efficiency of CPU usage, especially in the *webserver.f* workload where, in the worst case, there was a reduction of 60%.

Nevertheless, it is important to highlight the low GPU utilization in the experiments (Table 5). GPU utilization was only up to 25% in the *webserver.f* benchmark and around half of this amount in the *fileserver.f* workload. Also, this GPU use is lowering with more threads, as expected, because in this case, more threads competing for disk access generate more delays at this level, consequently generating less encryption work. This means that there is potential margin for scalability at the GPU side, provided bottlenecks at the FS/storage layers are resolved.

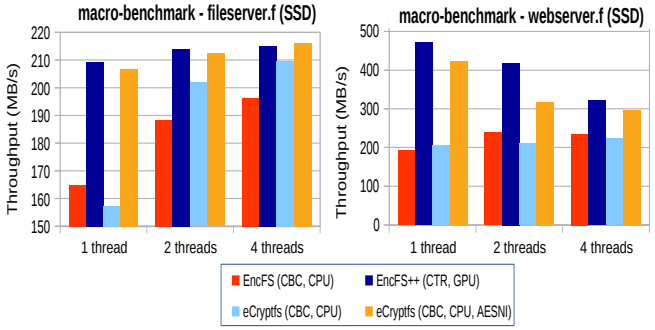

Figure 13: Macro-benchmarks (SSD).

Macro-benchmark – fileserver.f (SSD)																
Threads	EncFS (CBC, CPU)		eCryptfs (CBC, CPU)		eCryptfs (CBC, CPU, AESNI)		EncFS++ (CTR, GPU)			EncFS++/ EncFS(CBC)		EncFS++/ eCryptfs(CPU)		EncFS++/ eCryptfs(CPU,AESNI)	
	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	GPU (%)	Thrput Var. (%)	CPU use efficiency	Thrput Var. (%)	CPU use efficiency	Thrput Var. (%)	CPU use efficiency
1	164,8	9,9	157,2	8,8	206,9	4,6	209,4	4,7	10,5	27,1	2,7	33,2	2,5	1,2	1,0
2	188,2	11,4	201,9	11,5	212,4	4,4	214,0	5,8	12,8	13,7	2,2	6,0	2,1	0,8	0,8
4	196,2	12,3	209,7	16,1	216,1	10,9	215,1	5,8	11,4	9,6	2,3	2,6	2,9	-0,4	1,9
Macro-benchmark – webserver.f (SSD)																
Threads	EncFS (CBC, CPU)		eCryptfs (CBC, CPU)		eCryptfs (CBC, CPU, AESNI)		EncFS++ (CTR, GPU)			EncFS++/ EncFS(CBC)		EncFS++/ eCryptfs(CPU)		EncFS++/ eCryptfs(CPU,AESNI)	
	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	Thrput (MB/s)	CPU (%)	GPU (%)	Thrput Var. (%)	CPU use efficiency	Thrput Var. (%)	CPU use efficiency	Thrput Var. (%)	CPU use efficiency
1	192,3	8,0	204,9	10,9	422,5	3,6	471,8	11,2	25,5	145,4	1,8	130,3	2,2	11,7	0,4
2	239,7	10,8	211,1	11,2	316,1	3,6	418,3	10,5	23,3	74,5	1,8	98,2	2,1	32,3	0,5
4	234,8	10,0	222,9	11,7	296,2	7,0	322,3	8,8	19,1	37,3	1,6	44,6	1,9	8,8	0,9

Table 5: Macro-benchmarks: numbers (SSD).

7 Conclusion

CFSs have intensive processing demands due to the volume of data and the cost of cryptographic functions. Parallel processing techniques on CPUs or GPUs can be used to meet these demands. As a way to achieve better performance on these systems, you can use the parallel processing capabilities offered by multiprocessor computers and servers, whether in the form of multiple CPUs or GPUs.

In the application of the encryption functions, different modes of operation can be used, among them the CTR mode. It has interesting advantages, including the ability to be fully parallelizable and allowing the generation of encryption masks in advance. This work sought to explore both, including the anticipated generation of encryption masks in the GPU. The authors of this work are unaware of previous works that have explored them in the context of CFSs, which makes our approach innovative. While this work applies such techniques with the use of GPUs for parallel encryption tasks, they could also be exploited using CPUs only or in heterogeneous (CPU+GPU) solutions.

In this work, techniques were proposed to overcome two important challenges. The first concerns the use of the CTR

mode in the context of CFSs. Issues related to the management of nonces were addressed, including techniques for their generation and storage. The main objective was to ensure that the management of the nonces did not cause a significant decrease in the performance of the CFS, which could nullify the later benefits of parallel encryption tasks.

Future work involves macro-benchmark analysis using real workloads primarily to better evaluate the performance of context pools. This will allow to identify the need for improvements in management techniques created, or even the creation of new techniques. There is also the intention to apply the approach carried out in this work on storage systems that run in the kernel space, since these systems also usually demand greater processing capacity and we can circumvent extra delays introduced by user space modules.

Acknowledgement

We thank the anonymous reviewers, in particular we appreciate the help from our shepherd, Swami Sundararaman, for their many suggestions in improving this paper.

References

[1] A. D. Biagio, A. Barenghi, G. Agosta, and G. Pelosi. Design of a parallel AES for graphics hardware using the CUDA framework. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, May 2009.

[2] Morris J. Dworkin. SP 800-38A 2001 Edition. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2001.

[3] Morris J. Dworkin. Special Publication (SP) 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2007.

[4] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010.

[5] A. Gharaibeh and S. Al-Kiswany. Crystalgpu: Transparent and efficient utilization of gpu power. *arXiv preprint arXiv:1005.1695*, 2010.

[6] Johannes Gilger, Johannes Barnickel, and Ulrike Meyer. GPU-Acceleration of Block Ciphers in the OpenSSL Cryptographic Library. In *Proceedings of the 15th International Conference on Information Security*, ISC'12, pages 338–353, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] Valient Gough. EncFS: an Encrypted Filesystem for FUSE. https://github.com/vgough/encfs, note = Accessed 07/03/2017,.

[8] Michael Austin Halcrow. eCryptfs: An enterprise-class encrypted filesystem for linux. In *Proceedings of the 2005 Linux Symposium*, pages 201–218, 2005.

[9] Owen Harrison and John Waldron. GPU Accelerated Cryptography as an OS Service. In *Transactions on Computational Science XI*, pages 104–130. Springer-Verlag, Berlin, Heidelberg, 2010.

[10] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class GPU Resource Management in the Operating System. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.

[11] A. Keromytis, J. L. Wright, and T. Raadt. The Design of the OpenBSD Cryptographic Framework. In *USENIX Annual Technical Conference (General Track)*, pages 181–196, San Antonio, 2003. USENIX Association.

[12] Wai-Kong Lee, Hon-Sang Cheong, Raphael C.-W. Phan, and Bok-Min Goi. Fast Implementation of Block Ciphers and PRNGs in Maxwell GPU Architecture. *Cluster Computing*, 19(1):335–347, March 2016.

[13] Q. Li, C. Zhong, K. Zhao, X. Mei, and X. Chu. Implementation and Analysis of AES Encryption on GPU. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 843–848, June 2012.

[14] Lin, Shang-Chieh and Liao, Yu-Cheng and Hsu, Yarsun. A Reliable and Secure GPU-Assisted File System. In Sun, Xian-he and Qu, Wenyu and Stojmenovic, Ivan and Zhou, Wanlei and Li, Zhiyang and Guo, Hua and Min, Geyong and Yang, Tingting and Wu, Yulei and Liu, Lei, editor, *Algorithms and Architectures for Parallel Processing*, pages 71–84, Cham, 2014. Springer International Publishing.

[15] Helger Lipmaa, David Wagner, and Phillip Rogaway. Comments to NIST concerning AES modes of operation: CTR-mode encryption, 2000.

[16] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[17] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53, March 2008.

[18] Naoki Nishikawa, Keisuke Iwai, and Takakazu Kurokawa. High-Performance Symmetric Block Ciphers on Multicore CPU and GPUs. *International Journal of Networking and Computing*, 2(2):251–268, 2012.

[19] Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. Fast Software AES Encryption. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption*, pages 75–93, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[20] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[21] Margara Paolo. Engine-CUDA. https://github.com/heipei/engine-cuda, 2017. Accessed 06/05/2017.

[22] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, August 2003.

[23] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2014.

[24] Weibin Sun. *Harnessing GPU Computing in System-Level Software*. PhD thesis, School of Computing, University of Utah, Utah - EUA, 2014.

[25] Weibin Sun, Robert Ricci, and Matthew L. Curry. GPUstore: Harnessing GPU Computing for Storage Systems in the OS Kernel. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 9:1–9:12, New York, NY, USA, 2012. ACM.

[26] Chien-Kai Tseng, Shang-Chieh Lin, and Yarsun Hsu. A File System Using GPU-Accelerated File-wise Reliability Scheme. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 12)*, pages 32–38, Las Vegas, 2012. CSREA Press.

[27] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of user-space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, Santa Clara, CA, 2017.

[28] W. M. Nunan Zola and L. C. E. De Bona. Parallel speculative encryption of multiple AES contexts on GPUs. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, May 2012.