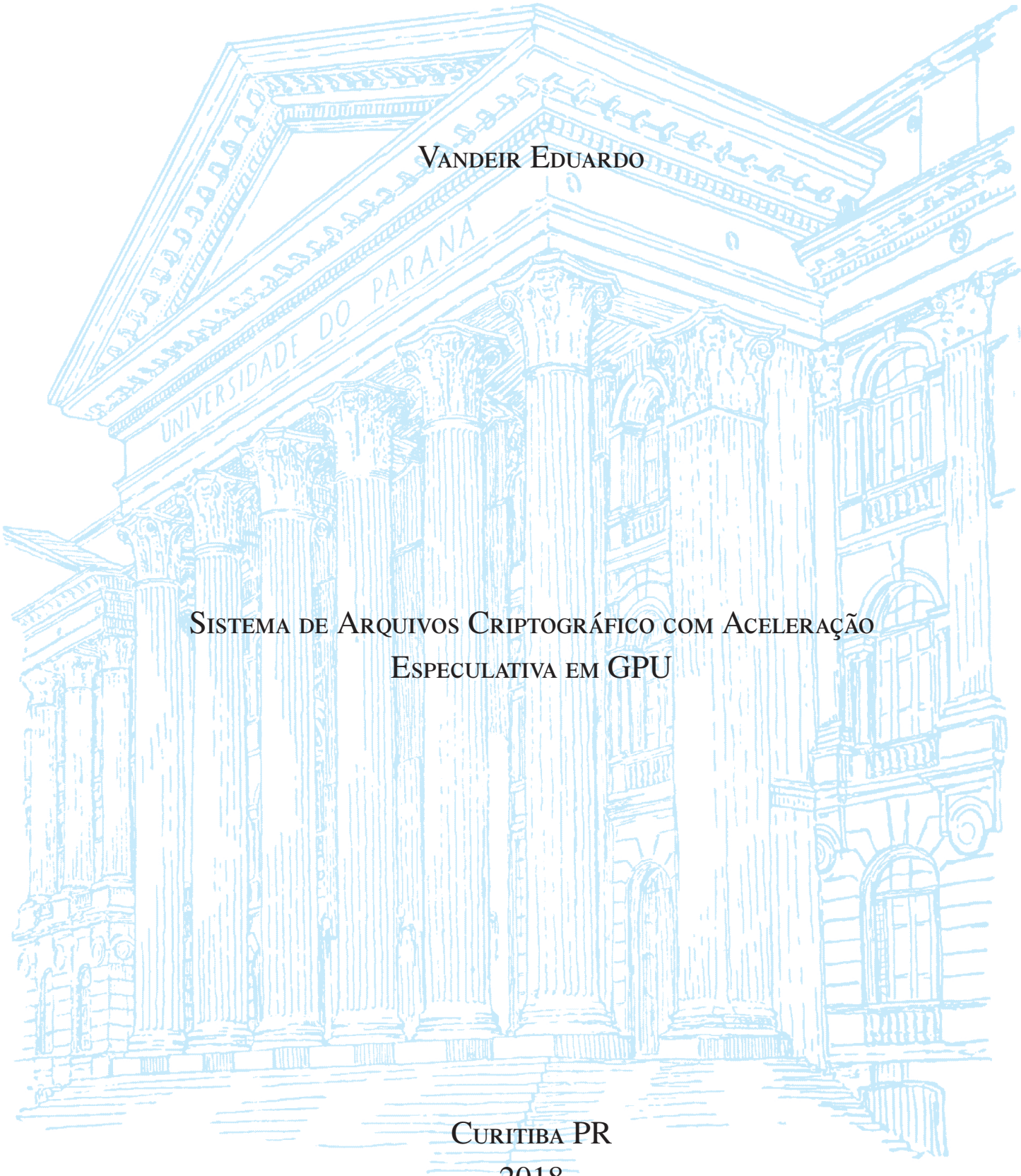
Universidade Federal do Paran´a



Vandeir Eduardo

Sistema de Arquivos Criptográfico com Aceleração Especulativa em GPU

Curitiba PR
2018

Vandeir Eduardo

Sistema de Arquivos Criptográfico com Aceleração Especulativa em GPU

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática, no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Dr. Wagner Machado Nunan Zola.

Coorientador: Dr. Luis Carlos Erpen de Bona.

Curitiba PR
2018

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

E24s Eduardo, Vandeir
Sistema de arquivos criptográfico com aceleração especulativa em GPU / Vandeir Eduardo. – Curitiba, 2018.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2018.

Orientador: Wagner Machado Nunan Zola – Coorientador: Luis Carlos Erpen de Bona. -

1. Criptografia 2. Processamento paralelo (Computadores) 3. Cifragem especulativa 4. Arquivos criptográfico. I. Universidade Federal do Paraná. II. Zola, Wagner Machado Nunan. III. Título.

CDD: 005.8

Bibliotecária: Vanusa Maciel - CRB - 9/1928

MINISTÉRIO DA EDUCAÇÃO
SETOR SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **VANDEIR EDUARDO** intitulada: **Sistema de Arquivos Criptográfico com Aceleração Especulativa em GPU**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua aprovação no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 03 de Setembro de 2018.

WAGNER MACHADO NUNAN ZOLA
Presidente da Banca Examinadora (UFPR)

WAGNER MEIRA JUNIOR
Avaliador Externo (UFMG)

ANDRÉ RICARDO ABED GRÉGIO
Avaliador Interno (UFPR)

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação da Informática

A minha amada mãe, Maria.

Agradecimentos

Gostaria de iniciar agradecendo a Deus por ter me dado o dom de sentir prazer ao trilhar o caminho pela busca de novos conhecimentos. Sem isto, mesmo com capacidade intelectual, o processo seria bem mais difícil. Na sequência, porém não menos importante, a minha família, com destaque merecido a minha amada mãe, dona Maria. Nas horas mais difíceis, sua palavras de conforto e incentivo fizeram toda a diferença para que eu não desistisse.

Obrigado também à família Menon, em especial a dona Vera e Rosália. Foram elas que me acolheram em Curitiba no primeiro ano de estudos e fizeram com que eu me sentisse em casa. Sua companhia, conversa, palavras de apoio e quitutes - sempre deliciosos - formavam a combinação perfeita para me ajudar a relaxar e recuperar as forças.

Com relação aos docentes, gostaria de agradecer ao professor Edgard Jamhour, cujas palavras ajudaram a reacender em mim a vontade de cursar um mestrado. Também aos meus orientadores, professores Wagner Zola e Luis Bona, pelas suas preciosas contribuições. Agradeço especialmente ao professor Wagner que claramente não poupou esforços para me ajudar, dedicando seu tempo inclusive aos finais de semana e em diversas madrugadas. Com certeza, sem sua dedicação, este trabalho não teria sido concluído.

Agradeço à Universidade de Blumenau (FURB), a qual ofereceu o suporte financeiro e logístico para que eu conseguisse conciliar meu trabalho e os estudos. Também aos meus colegas de trabalho, Dângelo e Charles, por terem segurado as pontas enquanto estive fora e me auxiliavam nas tarefas remotas quando necessário.

Por fim, gostaria de agradecer às demais pessoas que de alguma forma contribuíram para que eu conseguisse concretizar este trabalho, até mesmo àquelas que ajudaram com um simples, porém poderoso, pensamento positivo.

Obrigado de coração!

Resumo

A informação pode assumir um caráter valioso em diversas situações, inclusive ao ser armazenada em formato digital. É comum encontrar diversos sistemas de armazenamento de dados que se preocupam em cumprir com algumas propriedades básicas da segurança da informação. Geralmente utilizam técnicas de criptografia, principalmente a da cifragem simétrica. A utilização de criptografia pode exigir quantidades significativas de processamento em CPUs. Consequentemente, sistemas de armazenamento criptográficos podem se tornar grandes consumidores de recursos de processamento e ser impactados por outras aplicações ao concorrer pelo uso da CPU. Uma forma alternativa ao processamento em CPUs é o processamento paralelo utilizando múltiplos processadores de placas gráficas (GPUs). Um dos algoritmos de cifragem simétrica mais utilizados é o AES e sua aceleração em GPUs foi amplamente estudada. Um desses estudos resultou na criação do WAES e de sua biblioteca WAESlib, que permite executar funções de cifragem do AES em GPUs. O funcionamento do WAES está baseado no modo de operação CTR, o qual consiste em regras que orientam como devem ser aplicados os algoritmos de cifragem visando manter o processo de cifragem seguro. As principais vantagens do modo CTR são ser totalmente paralelizável e permitir realizar a etapa inicial do processo de cifragem de forma antecipada, gerando máscaras de cifragem. Procurando se beneficiar dessas vantagens, este trabalho explora a utilização do modo CTR, aplicando-o na implementação do sistema de arquivos criptográfico EncFS++. A biblioteca WAESlib foi utilizada para auxiliar no processo de implementação. Na primeira etapa deste trabalho foi implementado o modo CTR, onde foram tratadas questões relacionadas a um componente essencial do modo CTR denominado *nonce*. Foram criadas e implementadas técnicas que lidam com a geração, armazenamento e gerenciamento de *nonces*. Na segunda etapa foram criadas e implementadas técnicas relacionadas ao gerenciamento dos contextos de cifragem, procurando realizar a cifragem especulativa de forma eficiente, gerando as máscaras de cifragem na GPU com o tempo de antecedência adequado. Foram realizadas análises de desempenho envolvendo vazão, tempo de execução e latência na implementação resultante da primeira etapa, bem como vazão e utilização de CPU na implementação da segunda. Os resultados da primeira etapa demonstram que a simples utilização do modo CTR traz ganhos significativos de desempenho principalmente nas operações de escrita. Os resultados da segunda etapa demonstram que os ganhos podem ser ampliados, inclusive nas operações de leitura sequencial, com a produção especulativa das máscaras de cifragem e seu processamento em GPU. Em ambientes que não utilizam processadores com aceleração das funções criptográficas do AES, os ganhos são bem significativos, inclusive resultando em utilização mais eficiente da CPU.

Palavras-chave: sistema de arquivos criptográfico, criptografia, cifragem especulativa, processamento paralelo, GPU, CTR, EncFS, WAES, WAESlib.

Abstract

Information can be valuable in many situations, including when is stored in digital format. It is common to find several storage systems that try to comply with some basic information security properties. For those purposes, they use cryptographic techniques, mainly symmetric encryption. The use of cryptography may require significant amounts of processing on CPUs. As a result, cryptographic storage systems can become large consumers of processing resources and be impacted by other applications when competing for CPU usage. An alternative to CPU processing is parallel processing using multiple graphics processing units (GPUs). One of the most widely used symmetric encryption algorithms is AES and its acceleration in GPUs has been extensively studied. One of these studies resulted in the creation of WAES and its library named WAESlib, which allows execution of AES encryption functions on GPUs. The operation of WAES is based on CTR operation mode, which consists of rules that guide how encryption algorithms should be applied in order to keep the encryption process safe.The main advantages of CTR mode are to be fully parallelizable and allow to carry out the initial step of the encryption process in advance, generating encryption masks. In order to benefit from these features, this work explores the use of CTR mode, applying it in the implementation of a cryptographic filesystem named EncFS++. The WAESlib library was used to aid in the implementation process. In the first part of this work, CTR mode was implemented and issues related to an essential component of CTR mode known as nonce were addressed. Techniques have been created and implemented to deal with the generation, storage and management of nonces. In the second part, techniques related to the management of the encryption contexts have been created and implemented, aiming to perform the speculative encryption in an efficient way, generating the encryption masks in the GPU with adequate time in advance. Performance analysis were conducted measuring throughput, execution time and latency in the implementation resulting from the first part, as well as throughput and CPU utilization in the implementation of the second one. The performance analysis results of the first part demonstrate that the simple use of CTR mode brings significant performance gains, mainly in write operations. The performance analysis results of the second part demonstrate that gains can be enhanced, including in sequential read operations, with the speculative encryption of masks and its processing in GPU. In environments that do not use processors with accelerated AES cryptographic functions, gains in throughput were quite significant and a more efficient CPU utilization were obtained.

Keywords: cryptographic filesystem, cryptography, speculative encryption, parallel processing, GPU, CTR, EncFS, WAES, WAESlib.

Lista de Figuras

Lista de Tabelas

Lista de Quadros

Lista de Acrônimos

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
AES-NI	Advanced Encryption Standard New Instructions
AMD	Advanced Micro Devices
API	Application Program Interface
APU	Accelerated Processing Unit
CBC	Cipher Block Chaining
CC	Capacidade de Computação
CFB	Cipher Feedback
CFS	Cryptographic File System
CPU	Central Processing Unit
CRS	Cauchy Reed-Solomon
CTR	Counter
DES	Data Encryption Standard
DP	Double Precision
ECB	Electronic Codebook
E/S	Entrada/Saída
EFS	Encrypion File System
FIPS	Federal Information Processing Standards
FUSE	File System in User Space
GCM	Galois Counter Mode
GCN	Graphics Core Next
GDDR5	Double Data Rate type five synchronous Graphics Random-access memory
GPGPU	General-purpose Programming using Graphics Processing Unit
GPU	Graphics Processing Unit
HBM2	High Bandwidth Memory, second generation
HMAC	keyed-Hash Message Authentication Code
I/O	Input/Output
IV	Initialization Vector
LD/ST	Load/Store
LUKS	Linux Unified Key Setup
LVM	Logical Volume Manager
MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data
MSB	Most Significant Bits
NESSIE	New European Schemes for Signatures, Integrity and Encryption
NFS	Network File System
NIST	National Institute of Standards and Technology

OCF	OpenBSD Cryptographic Framework
OFB	Output Feedback
OpenCL	Open Computing Language
PAM	Pluggable Authentication Module
PBKDF2	Passwork-Based Key Derivation Function 2
PCIe	Peripheral Component Interconnect Express
PKY	Public Key Infrastructure
RAID	Redundant Array of Independent Disks
RAM	Random-Access Memory
RPC	Remote Procedure Call
SAB	Sistema de Arquivos de Base
SAC	Sistema de Arquivos Criptográfico
SFU	Special Function Unit
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SISD	Single Instruction Single Data
SM	Streamming Multiprocessor
SO	Sistema Operacional
SP	Special Publication
SPN	Substitution-Permutation Network
SSD	Solid State Disk
TCFS	Transparent Cryptographic File System
TPM	Trusted Platform Module
VFS	Virtual File System
WAES	Warped AES
XTS	XEX-based Tweakable codebook mode with ciphertext Stealing

Sumário

Capítulo 1

Introdução

A necessidade de se manter a informação segura é comum em diversos contextos. Não é algo recente e remonta à época de civilizações antigas. Essa necessidade levou a criação de técnicas que visam tornar a informação sigilosa e compreensível somente às partes autorizadas. A criptografia engloba várias dessas técnicas. Inicialmente seu estudo era mais restrito aos campos militar e governamental. Porém, a partir da década de 70, houve a transição para a academia. Esse foi o início do processo que culminaria no desenvolvimento e disseminação que a criptografia possui atualmente [1].

A importância e necessidade de se manter segura a informação também fez surgir e se desenvolver outra área de estudo dedicada a esse tópico, chamada de segurança da informação. Nela, estão definidas três propriedades principais: confidencialidade, integridade e disponibilidade [2]. Ao se armazenar dados de forma segura, é necessário empregar mecanismos visando cumprir os requisitos associados a algumas ou todas essas propriedades. Uma forma de se alcançar tal objetivo é através da utilização das técnicas de criptografia.

Principalmente devido a sua característica de tempo de processamento, a cifragem simétrica, uma das formas de aplicação da criptografia, é uma das principais técnicas utilizadas para cifrar dados quando esses são armazenados [2]. Mais especificamente, são utilizados cifradores simétricos de blocos. Existem regras específicas que orientam como eles devem ser utilizados ao se dividir os dados em blocos e realizar a sua cifragem ou decifragem. Elas visam manter a segurança do processo, procurando evitar ou reduzir ao máximo as chances de sucesso de ataques conhecidos. Essas regras são conhecidas como modos de operação.

Inicialmente os recursos de criptografia eram restritos a determinadas aplicações, sendo necessário usar ferramentas separadas para tornar segura a informação armazenada. Considerando que os dados, ao serem armazenados, normalmente são organizados em pastas e arquivos, sendo estes controlados pelos sistemas de arquivos, foi natural ocorrer a integração dos recursos de criptografia nesses sistemas. Dessa abordagem surgiram os sistemas de arquivos criptográficos (SACs).

Os SACs, ao utilizarem as técnicas de cifragem simétrica, realizam diversas rodadas de operações sobre os dados. Operações de leitura e gravação podem resultar numa quantidade considerável de dados a serem cifrados ou decifrados. Como consequência, tem-se um aumento no uso dos recursos de processamento da *Central Processing Unit* (CPU). Visando reduzir a utilização de recursos da CPU e diminuir o tempo de processamento, foram criadas soluções combinadas de hardware e software, chamadas placas aceleradoras, cuja principal função é realizar o processamento das funções que envolvem criptografia. Porém, essas placas tinham um alto custo e pouca flexibilidade [3].

Outra opção de placas aceleradoras são as placas de vídeo e suas unidades de processamento chamadas *Graphics Processing Units* (GPUs). Inicialmente eram exclusivamente utilizadas para acelerar o processamento de funções gráficas, utilizando linguagens específicas para essa finalidade. Posteriormente, a fabricante de placas de vídeo NVIDIA lançou a plataforma CUDA, criando um ambiente mais natural para criar aplicações que utilizassem os recursos das GPUs para realizar diversos tipos de processamento. A partir desse ponto, tornou-se comum a criação e adaptação de vários algoritmos para explorar os recursos de processamento paralelo oferecidos pelas GPUs, incluindo os algoritmos de criptografia.

A aceleração do processamento em GPUs do cifrador simétrico Rijndael, conhecido como *Advanced Encryption Standard* (AES), padrão oficializado pelo *National Institute of Standards and Technology* (NIST), também foi bastante estudada [4][5][6][7][8][9], incluindo sua aplicação em SACs [10][11][12][13].

Um dos estudos relacionados à aceleração do AES em GPUs resultou no *Warped* AES (WAES), apresentando ganhos significativos de desempenho [14]. Esse estudo também resultou na criação da biblioteca WAESlib, a qual pode ser utilizada para facilitar a integração de aplicações com o uso de processamento criptográfico do AES em GPUs.

Um grande diferencial do WAES, além do processamento em GPU, é a utilização do modo de operação denominado *Counter* (CTR). O modo CTR tem duas características particularmente úteis: a capacidade de ser totalmente paralelizável em ambas as operações de cifragem e decifragem e a capacidade de realizar a etapa inicial de cifragem ou decifragem de forma antecipada, gerando o que convencionamos chamar de máscaras de cifragem.

Além de explorar a primeira característica, o WAES também explora a segunda, tornando-se capaz de computar dados cifrados de forma antecipada. Assim, quando uma aplicação efetivamente precisar de dados cifrados, eles já estarão disponíveis, prontos para serem utilizados. Essa característica pode se mostrar útil em várias situações, incluindo tornar efetiva a cifragem em GPU de pequenas quantidades de dados. Como boa parte das operações de sistemas de arquivos podem ficar abaixo ou na casa dos 4 KiB, a utilização do WAES pode trazer resultados interessantes.

Porém, dotar um SAC da capacidade de processar suas funções criptográficas em GPU não se restringe a simples utilização de uma biblioteca. Para que esse processamento seja feito de forma eficiente, há pelo menos dois desafios significativos a serem vencidos.

O primeiro diz respeito a como implementar o modo CTR em um SAC respeitando os requisitos de segurança exigidos pelo modo. É necessário tratar questões relacionadas a um elemento essencial ao funcionamento do modo CTR denominado *nonce*. Essas questões dizem respeito a sua geração, armazenamento e gerenciamento. A implementação resultante precisa garantir que a utilização do modo CTR não cause impactos negativos ao desempenho do SAC, os quais poderiam anular os ganhos advindos do processamento das funções criptográficas em GPU.

O segundo está diretamente relacionado ao processamento em GPU e se refere a como gerenciar os contextos de cifragem, utilizados pela biblioteca WAESlib, para controlar a produção das máscaras de cifragem na GPU e sua posterior utilização nos processos de cifragem e decifragem de dados. As operações de leitura e escrita, tanto sequenciais quanto aleatórias, possuem características distintas, o que exige a criação de diferentes técnicas de gerenciamento desses contextos visando produzir as máscaras de cifragem com o tempo de antecedência adequado.

Técnicas de como superar esses desafios, bem como formas de implementá-las, constituem o cerne deste trabalho e estão descritas nos Capítulos 5 e 6.

1.1 Objetivos

Levando em conta o cenário anteriormente descrito, parece bastante promissor explorar as vantagens do modo CTR no contexto de SACs, incluindo a geração antecipada de máscaras de cifragem em GPU. A finalidade principal desse processo é melhorar o desempenho do SAC, principalmente procurando conseguir maiores taxas de vazão e uma utilização mais eficiente dos recursos de processamento da CPU.

Neste sentido, este trabalho propõe a implementação do modo CTR sobre um SAC denominado EncFS, com posterior aperfeiçoamento de seu desempenho através do processamento de suas funções criptográficas em GPU. Os principais objetivos são justamente explorar os recursos de processamento paralelo oferecido pelo modo CTR, bem como o cômputo antecipado de máscaras de cifragem através de sua geração de forma especulativa. Como forma de auxiliar neste processo, é utilizado o WAES e sua biblioteca WAESlib.

1.2 Contribuições

Como resultado geral, as principais contribuições deste trabalho são:

- Uma revisão bibliográfica na área de sistemas de armazenamento criptográfico baseados em software, bem como da aceleração de funções criptográficas em GPU e sua aplicação nestes tipos de sistemas de armazenamento;
- Demonstrar como o modo CTR pode ser aplicado no contexto de SACs e como sua simples utilização resulta em ganhos de desempenho em alguns cenários. Neste sentido, são apresentadas técnicas relacionadas principalmente à geração, armazenamento e gerenciamento de *nonces*;
- Demonstrar como os ganhos de desempenho provenientes da utilização do modo CTR podem ser ampliados com o uso da cifragem especulativa em GPU. São apresentadas técnicas de como gerenciar os contextos de cifragem, criando *pools* de contextos, de tal forma que seja possível gerar as máscaras de cifragem com a antecedência adequada;
- Validação das ideias de utilização do modo CTR e da cifragem especulativa em GPU através de sua implementação em um SAC que passamos a chamar de EncFS++;
- Validação dos resultados através de análises de desempenho analisando quesitos como vazão, tempo de execução, latência e utilização de CPU em diversos cenários.

Além das contribuições acima descritas, uma parte dos resultados obtidos neste trabalho foi publicada e apresentada no XVI *Workshop* em *Clouds* e Aplicações (WCGA - SBRC 2018) [15].

1.3 Organização

O restante do trabalho está organizado da seguinte forma:

- No Capítulo 2 são apresentados conceitos relacionados à criptografia, com foco em cifragem simétrica. Na sequência, são apresentados alguns detalhes do algoritmo de cifragem simétrica AES e os modos de operação que podem ser utilizados nos processos de cifragem;

- No Capítulo 3 são vistos sistemas de armazenamento com recursos de criptografia. São apresentados sistemas que exemplificam os diferentes níveis nos quais eles podem atuar cifrando ou decifrando dados;
- O Capítulo 4 apresenta a arquitetura de GPUs NVIDIA e como os recursos de GPUs podem ser utilizados através da plataforma CUDA. Os Capítulos 2, 3 e 4 constituem a fundamentação teórica deste trabalho;
- No Capítulo 5 são vistas questões relacionadas à implementação do modo CTR no SAC EncFS, onde são aplicadas as técnicas ligadas à geração, armazenamento e gerenciamento de *nonces*. São apresentadas análises de desempenho da implementação resultante;
- No Capítulo 6 são vistos assuntos relacionados à aplicação da cifragem especulativa em GPU, incluindo os mecanismos criados para gerenciamento dos contextos de cifragem, com o auxílio da biblioteca WAESlib. Também são apresentadas análises de desempenho que avaliam os resultados obtidos com o processamento das funções criptográficas em GPU;
- O Capítulo 7 apresenta um resumo e considerações gerais acerca dos resultados obtidos, bem como propõe trabalhos futuros que permitem estender e aperfeiçoar este trabalho.

Capítulo 2

Cifragem simétrica e modos de operação

A utilização de técnicas de criptografia não é mais algo aplicado exclusivamente para garantir a segurança de comunicações governamentais, principalmente militares, ou sistemas bancários. Atualmente, essas técnicas são utilizadas nos mais variados sistemas computacionais, estando presentes em boa parte das aplicações utilizadas no dia a dia. São aplicações como navegadores de internet, leitores de e-mail, telefones celulares, sistemas de armazenamento de dados, cartões de banco, etc.

Criptografia é o estudo de técnicas matemáticas que visam garantir algumas propriedades da segurança da informação. Entre elas: (i) Confidencialidade: diz respeito a manter a informação confidencial, ou seja, acessível ou compreensível somente às partes autorizadas; (ii) Integridade: refere-se a garantir que não seja possível, ou, pelo menos, que se possa detectar alterações no conteúdo da informação; (iii) Autenticidade: é o processo de identificação das partes envolvidas no processo de troca ou acesso à informação, de tal forma que seja possível verificar quem as partes afirmam ser; (iv) Irretratabilidade: é uma forma de garantir que uma parte não seja capaz de negar que realizou o acesso ou envio de uma determinada informação [2].

A criptografia se divide em três partes principais: (i) Algoritmos de cifragem[1] simétrica: utilizados para cifrar o conteúdo de blocos ou fluxo de dados, onde as partes envolvidas compartilham a mesma chave; (ii) Algoritmos de cifragem assimétrica: onde cada parte envolvida na troca da informação possui um par de chaves, uma sendo privada e a outra pública; (iii) Protocolos de cifragem: são os protocolos utilizados pelas aplicações, os quais fazem uso, na prática, dos algoritmos de cifragem [2].

Entende-se por cifragem a utilização de técnicas específicas que visam tornar o conteúdo de uma determinada informação incompreensível a quem desconhece a técnica e/ou chave de cifragem utilizada. Já decifragem é o processo inverso: a utilização de técnicas que visam reverter o processo de cifragem a fim de que se possa obter novamente a informação original e compreensível.

Um dos principais objetivos de sistemas de armazenamento criptográficos é garantir a confidencialidade dos dados. São caracterizados por trabalharem com grandes quantidade de dados e, por essa razão, necessitam que o processo de cifragem seja rápido, não consumindo muitos recursos de processamento.

Duas características associadas a algoritmos de cifragem simétrica são a capacidade de atingirem altas taxas de vazão de dados quando em operação e também não necessitarem de chaves muito grandes para se obter níveis suficientes de segurança [2]. Por isso, os sistemas de armazenamento criptográficos fazem uso intensivo da cifragem simétrica.

[1]Será dada preferência a utilização dos termos cifrar e decifrar, porém é comum encontrar na literatura especializada o uso dos termos encriptar e decriptar, os quais possuem o mesmo significado, respectivamente.

Na Seção 2.1 são descritas algumas características da cifragem simétrica e em blocos. Na Seção 2.2 é apresentada uma visão da cifra simétrica de bloco conhecida como *Advanced Encription Standard* (AES). Na Seção 2.3 são vistos cinco modos básicos de como as cifras de bloco podem ser aplicadas quando realizam a cifragem de dados. Por fim, a Seção 2.4 apresenta uma visão geral de outros cifradores simétricos e modos de operação.

2.1 Cifragem simétrica

A cifragem simétrica consiste no processo de cifragem da informação utilizando a mesma chave que será utilizada no processo de decifragem. Um modelo simplificado de cifragem simétrica pode ser visto na Figura 2.1. O termo "texto claro" se refere à informação em seu formato original antes de ser submetida ao processo de cifragem. O termo "Algoritmo de cifragem" se refere ao algoritmo que executará as operações necessárias sobre o texto claro a fim cifrá-lo. O termo "Chave secreta" se refere a chave que o algoritmo de cifragem utiliza para realizar o processo de cifragem. O termo "Texto cifrado" se refere ao texto gerado após a aplicação do algoritmo de cifragem. O termo "Algoritmo de decifragem" se refere ao algoritmo que fará a operação inversa, ou seja, a transformação do texto cifrado novamente em texto claro.

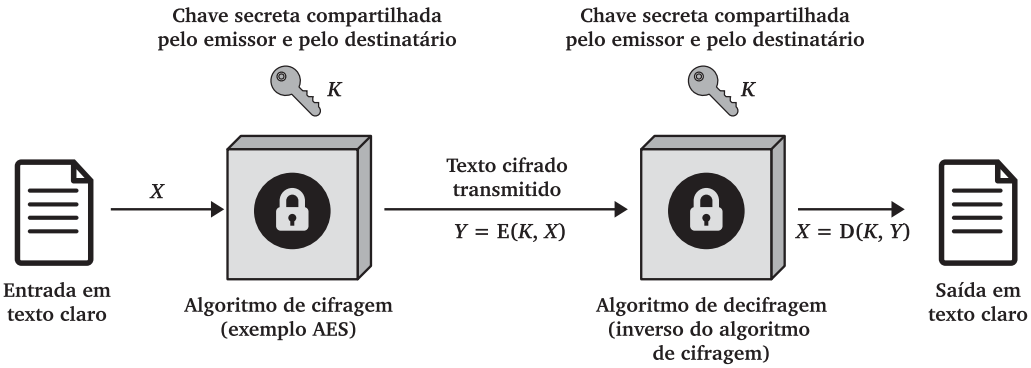


Figura 2.1: Modelo de cifragem simétrica. Adaptado de [16].

Nesse modelo, a entrada é uma mensagem em texto claro, $X = [X_1, X_2, \ldots, X_M]$. Para a cifragem e decifragem, uma chave no formato $K = [K_1, K_2, \ldots, K_J]$ também é gerada. Essa chave precisa ser compartilhada entre as partes previamente através de um canal seguro. Com a mensagem X e a chave de cifragem K como entrada, o algoritmo de cifragem gera o texto cifrado na forma $Y = [Y_1, Y_2, \ldots, Y_N]$. Isso pode ser representado pela função $Y = E(K, X)$. Essa notação indica que o texto cifrado Y é o resultado da aplicação da função de cifragem E, a qual tem como entrada o texto claro X e a chave K. O receptor da mensagem, para obter o texto claro, ou seja, decifrado, aplica a função $X = D(K, Y)$, onde D representa a aplicação do algoritmo de decifragem que tomará como entrada o texto cifrado Y e a chave K.

De acordo com a forma com que os dados são cifrados, as cifras simétricas podem ser classificadas em cifras de bloco e cifras de fluxo. Nas cifras de blocos, o processo de cifragem é realizado sobre um bloco de dados de tamanho fixo. Na maioria das cifras atuais, esses blocos possuem 64 e 128 bits. Nas cifras de fluxo, os bits são cifrados individualmente, produzindo um bit de cada vez na saída.

A Figura 2.2 ilustra o funcionamento da cifra de bloco. O texto claro consiste de um bloco com b bits de tamanho. Esse bloco, bem como a chave (K), servem com entrada ao algoritmo de cifragem, o qual gera como saída um bloco do mesmo tamanho do bloco de entrada,

contendo o texto cifrado. No receptor, para se obter novamente o bloco contendo o texto claro, o bloco contendo o texto cifrado é submetido ao algoritmo de decifragem utilizando a mesma chave K.

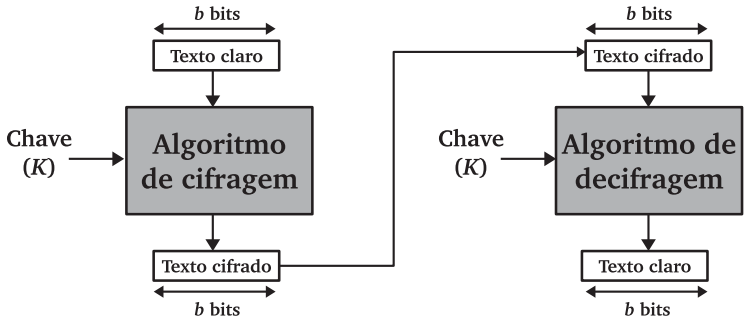


Figura 2.2: Funcionamento da cifra de bloco. Adaptado de [16].

As cifras de bloco simétricas utilizam duas técnicas em seu funcionamento: a substituição e a transposição. A substituição consiste no processo de trocar um símbolo[2] ou grupo de símbolos por outro símbolo ou grupo de símbolos diferentes. A transposição consiste na permutação das posições ocupadas por esses símbolos ou grupo de símbolos dentro do bloco de dados que será cifrado. Cifras que utilizam individualmente apenas técnicas de substituição ou transposição não são seguras. Isso porque essas técnicas utilizadas separadamente não conseguem esconder propriedades estatísticas do texto claro no texto cifrado [2][1][17].

Para prevenir ataque estatísticos, cifras seguras combinam as duas técnicas, sendo denominadas cifras de produto. Além disso, essas técnicas são aplicadas alternada e sucessivamente um determinado número de vezes. As sucessivas aplicações dessas técnicas de forma combinada recebem o nome de rodadas de cifragem [2]. A utilização da técnica de substituição numa determinada rodada visa adicionar confusão e a utilização da técnica de transposição adiciona difusão [2].

O principal objetivo da confusão é tornar a mais complexa possível qualquer relação que exista entre a chave e o texto cifrado. Já a difusão visa espalhar a influência de um determinado símbolo a ser cifrado sobre vários símbolos cifrados, procurado esconder propriedades estatísticas do texto claro. A utilização de ambas as técnicas, aplicadas sucessivamente em rodadas, constituem o mecanismo de funcionamento de boa parte das cifras simétricas atuais [1][17][18]. Essa ideia está ilustrada na Figura 2.3.

2.2 *Advanced Encryption Standard*

Em 1997, o Instituto Nacional de Padrões e Tecnologia dos Estados Unidos (NIST[3]) fez uma chamada pública com o objetivo de escolher um novo algoritmo de cifragem, o qual se tornaria o novo padrão de cifra simétrica a ser utilizada.

Um dos principais motivos dessa chamada era encontrar um novo algoritmo para substituir o *Data Encryption Standard* (DES) e o *triple* DES (3DES). Na época, já havia sido comprovada a possibilidade de ataques por força bruta ao DES. Apesar de ainda ser considerado seguro, o 3DES, assim com o DES, não apresenta características que o tornam rápido quando implementado em software.

[2]Entende-se por símbolo um elemento pertencente a um conjunto finito de elementos. Por exemplo letras de um alfabeto, um conjunto de números, etc.

[3]Sigla do inglês *National Institute of Standards and Technology*.

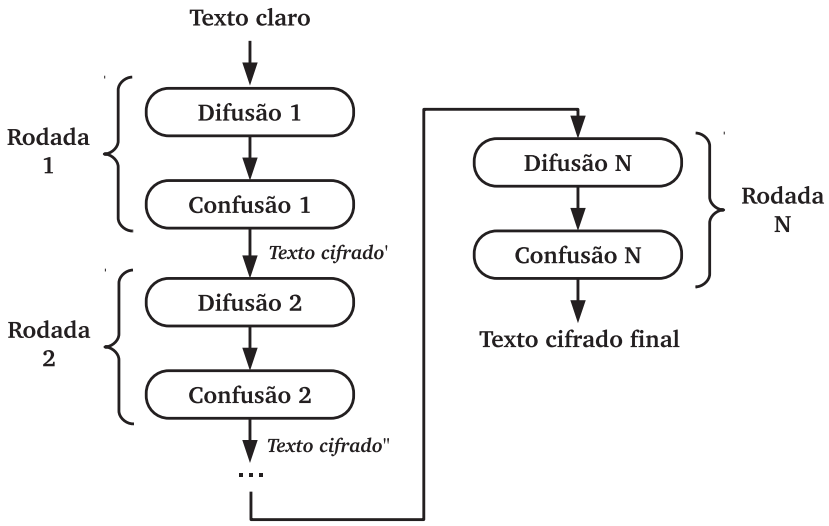


Figura 2.3: Aplicação da difusão e confusão em rodadas. Adaptado de [1].

Até 1998, 15 algoritmos foram submetidos à chamada e em 1999 cinco algoritmos foram declarados finalistas: Mars, RC6, Rijndael, Serpent e Twofish. Em 2000, o NIST declarou que o algoritmo Rijndael, desenvolvido por dois criptógrafos belgas[4], foi o vencedor e declarado o novo AES. Em 2001, o NIST publicou o padrão FIPS PUB[5] 197 [19], descrevendo o funcionamento do AES.

Seguindo as exigências do NIST, o AES trabalha com blocos de 128 bits (16 bytes) e com três tamanhos diferentes de chaves: 128, 192 e 256 bits (16, 24 e 32 bytes, respectivamente). De acordo com a especificação do padrão, esses 128 bits de entrada são dispostos numa matriz de 4x4. Cada byte ocupada uma posição da matriz e os mesmos são dispostos na matriz orientados pela coluna. Essa matriz é denominada de estado, sendo o mesmo alterado em cada estágio no processo de cifragem.

O tamanho da chave (128, 192 e 256 bits) determina a quantidade de rodadas que são aplicadas ao estado no processo de cifragem. São 10, 12 ou 14 rodadas. A chave utilizada passa por um algoritmo de expansão, e também de acordo com o tamanho dela, gera um vetor de 176, 208 ou 240 bytes. Esse vetor permite armazenar 11, 13 e 15 chaves, as quais são utilizadas em cada uma das rodadas do algoritmo.

Cada rodada, com exceção da última, consiste em quatro funções de transformação: substituição de bytes, deslocamento de linhas, mistura de colunas e adição da chave da rodada[6]. A função de mistura de colunas não é executada na última rodada. A função de substituição de bytes é responsável por aplicar o princípio da confusão sobre os dados. As funções de deslocamento de linhas e mistura de colunas aplicam o princípio da difusão.

O processo de decifragem segue uma sequência de operações semelhantes. Porém, utilizando funções diferenciadas das anteriormente citadas, as quais visam reverter o processo de cifragem. As chaves das rodadas são utilizadas na ordem inversa.

A Figura 2.4 ilustra o processo de cifragem e decifragem do AES no caso da utilização de uma chave de 128 bits. Na Figura 2.4 (a), do lado esquerdo e de cima para baixo, está ilustrada a sequência de operações do AES ao realizar o processo de cifragem de dados. Na Figura 2.4

[4]Joan Daemen e Vincent Rijmen.

[5]Sigla do inglês *Federal Information Processing Standards Publication*.

[6]Originalmente, no padrão FIPS PUB 197, as funções de cifragem são denominadas: SubBytes, ShiftRows, MixColumns. As funções de decifragem são denominadas: InvSubBytes, InvShiftRows e InvMixColumns. A função de adição da chave da rodada é denominada AddRoundKey.

(b), do lado direito e de baixo para cima, está ilustrada a sequência de operações para o processo de decifragem. Cada chave da rodada utilizada está descrita no formato $w[x, y]$, sendo w o vetor de chaves das rodadas, x a posição do *word* (4 bytes) inicial da chave no vetor e y o *word* final.

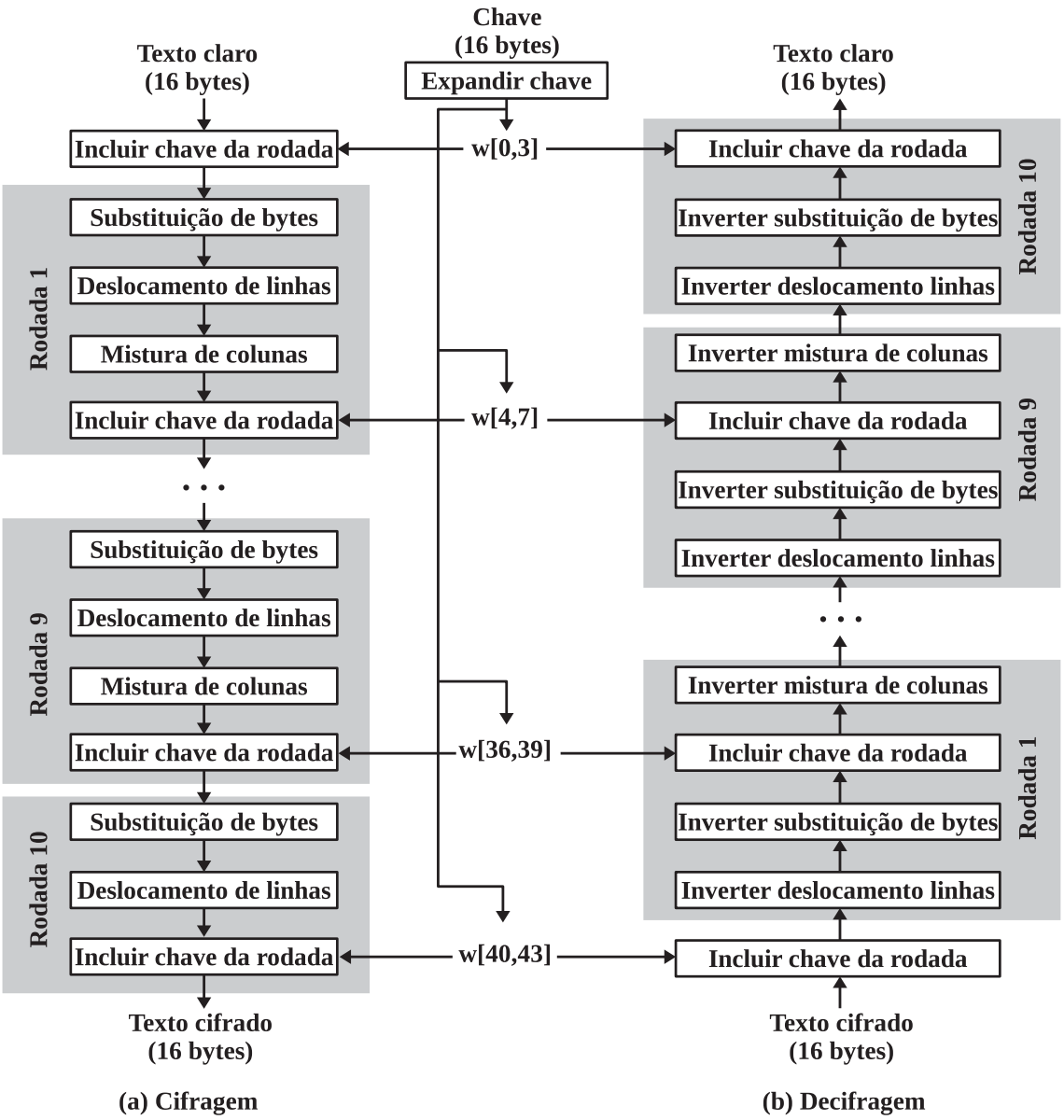


Figura 2.4: Visão geral das rodadas do AES para chave com 128 bits. Adaptado de [16].

2.3 Modos de operação para cifras de bloco

As cifras simétricas de bloco trabalham com blocos de tamanhos específicos, no caso do AES são blocos de 128 bits. Operações normais de cifragem sobre mensagens ou arquivos trabalham com quantidades maiores de dados. Uma forma óbvia de resolver esse problema é dividir a quantidade total de dados em blocos de tamanho correspondente ao tamanho do bloco da cifra e processá-los individualmente. Basicamente, é assim que funcionam os modos de operação para as cifras de bloco.

Porém, para garantir que nesse processo algumas propriedades de segurança sejam mantidas, esses modos de operação precisam ser implementados de uma forma específica. Com essa finalidade, o NIST lançou uma publicação especial denominada SP[7] 800-38A [20], onde há recomendações de como deve ser o funcionamento de cinco modos de operação diferentes. São os modos *Electronic Codebook* (ECB), *Cipher Block Chaining* (CBC), *Cipher Feedback* (CFB), *Output Feedback* (OFB) e *Counter* (CTR). As subseções seguintes, baseadas no livro [17] e na SP 800-38A, descrevem o funcionamento desses modos de operação.

2.3.1 Modo *Electronic Codebook* (ECB)

Esse é o modo mais simples onde os dados a serem cifrados são divididos em blocos de b bits, onde b corresponde ao tamanho em bits do bloco da cifra simétrica sendo utilizada. Cada bloco é cifrado independentemente um do outro utilizando a mesma chave. O processo de decifragem é aplicado de forma idêntica. Ambos podem ser vistos na Figura 2.5. Nela, $P_1, P_2, \ldots, P_N$ indicam os blocos de b bits correspondentes ao texto claro; K a chave sendo utilizada; $C_1, C_2, \ldots, C_N$ os blocos também de b bits de texto cifrado.

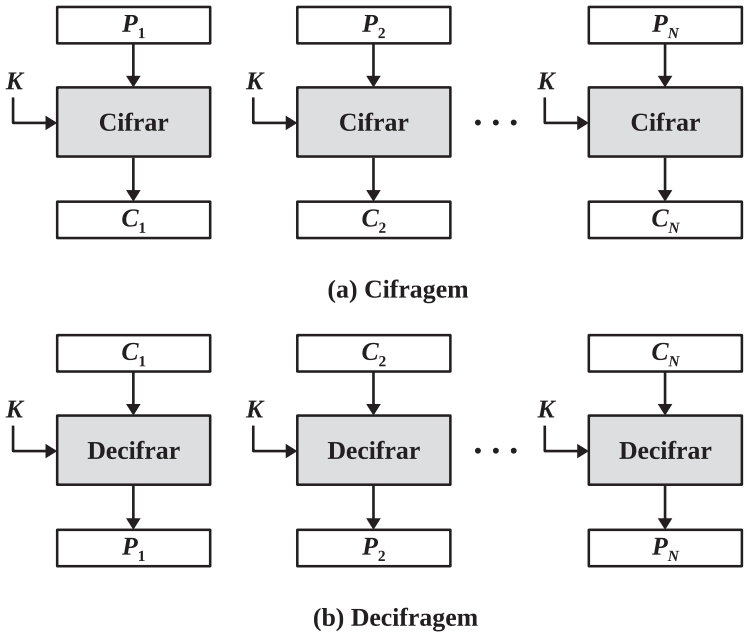


Figura 2.5: Modo de operação ECB. Adaptado de [16].

É importante observar que ao se dividir os dados a serem cifrados em blocos de b bits, essa divisão pode resultar num número não inteiro, o que ocorre quando o tamanho dos dados não for um múltiplo do tamanho do bloco. Nesse caso, para os modos ECB, CBC e CFB é necessário aplicar uma técnica chamada de *padding* para garantir que todos os blocos estejam totalmente preenchidos.

Em [21] e [20] são discutidas duas formas de implementar essa técnica de *padding*. A mais simples é adicionar um único byte com o valor decimal de 128 ao final do último bloco e preencher o restante com bytes zeros até completar o tamanho do bloco. A outra técnica consiste no seguinte: considerando o tamanho que o bloco precisa ter em bytes como sendo b, $l(P)$ o tamanho que o último bloco possui e n a quantidade de bytes necessários a serem adicionados, de tal forma que $1 \leq n \leq b$ e $n + l(P)$ seja um múltiplo de b, adiciona-se ao final do último bloco,

[7] Sigla do inglês *Special Publication*.

n bytes contendo o valor *n*. Ambas as formas permitem que o último bloco tenha o tamanho correto para ser processado pela cifra de bloco e também permitem reverter essa adição de bytes para que o *pad* seja descartado após o processo de decifragem.

Segundo [2], [17], [21] e [20] o modo ECB não é seguro na maioria dos casos. Isso se deve ao fato de que as únicas entradas no seu processo de funcionamento são o texto claro e a chave, não havendo nenhum elemento a mais que adicione aleatoriedade. Dessa forma, se dois blocos de texto claro iguais forem cifrados com a mesma chave, isso resulta em dois blocos cifrados idênticos. Essa característica torna o modo ECB vulnerável a algumas formas de ataque.

2.3.2 Modo *Cipher Block Chaining* (CBC)

Neste modo de operação, os blocos cifrados deixam de ser gerados em função apenas do bloco de texto claro e da chave. Nele, cada bloco cifrado depende do bloco de texto claro, da chave e do bloco de texto cifrado imediatamente anterior a ele. Antes de um bloco ser cifrado, ele é submetido a uma operação de `XOR` com o bloco cifrado anterior. Somente após essa operação ele é cifrado. É justamente essa forma de encadeamento com o bloco cifrado anteriormente que garante a aleatoriedade na geração dos blocos cifrados. Assim, mesmo que haja dois blocos de texto claro sendo cifrados com a mesma chave, ambos resultarão em blocos cifrados diferentes. Isso resolve o problema de segurança associado ao modo ECB anteriormente descrito.

O processo de decifragem é ligeiramente diferente. Primeiro cada bloco de texto cifrado é submetido ao processo de decifragem para depois ser submetido ao procedimento de `XOR` com o bloco de texto cifrado anterior. Assim, obtém-se os blocos de texto claro originais. Ambos os procedimentos de cifragem e decifragem podem ser vistos na Figura 2.6.

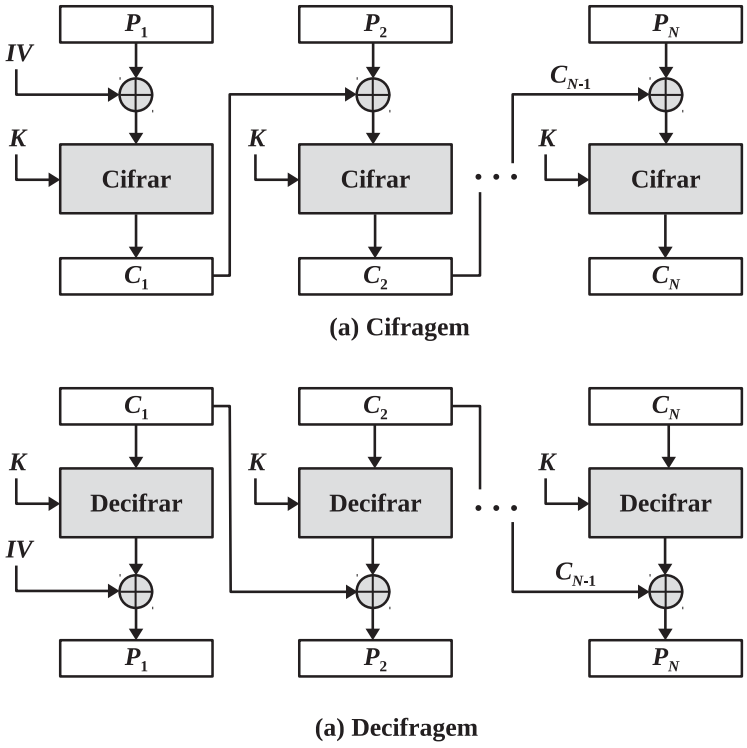


Figura 2.6: Modo de operação CBC. Adaptado de [16].

Também como pode ser visto na Figura 2.6, há um detalhe com relação ao primeiro bloco a ser processado. Como nesse caso não há um bloco anterior para ser utilizado, utiliza-se

um vetor de inicialização (IV[8]). O IV corresponde a um bloco de dados do mesmo tamanho dos demais blocos. A escolha dos dados a serem utilizados para popular esse IV deve seguir algumas regras de tal forma a manter a propriedade de confidencialidade do modo de operação.

Segundo [20], para os modos CBC e CFB a principal característica dos dados contidos no IV é que os mesmos não podem ser previsíveis. Ou seja, que não seja possível a um atacante, com base no texto claro, prever qual será o IV utilizado num processo futuro de cifragem. Não é obrigatório o IV ser mantido em segredo. Como ambos os processos de cifragem e decifragem precisam utilizar o mesmo IV, ambas as partes que se comunicam precisam ter conhecimento do IV utilizado. Portanto, o IV pode ser enviado no início da comunicação para que a outra parte saiba qual IV utilizar no processo de decifragem.

Segundo [1], [17], [21] e [20] o que geralmente se faz é utilizar um *nonce*[9], o qual é cifrado com a mesma chave utilizada no restante do processo de cifragem. O único requisito obrigatório é que esse *nonce* nunca seja reutilizado em outros processos de cifragem que utilizem a mesma chave. Segundo [20], uma outra forma de gerar o IV é utilizando um gerador de números aleatórios que seja aprovado pelo FIPS.

Na SP 800-38A original, no modo de operação CBC, o tamanho dos dados a serem processados nesse modo de operação também precisava ser múltiplo do tamanho do bloco. Caso contrário, seria necessária a aplicação da técnica de *padding*. Mais tarde, foi publicado um adendo descrevendo três variações do modo CBC utilizando uma técnica chamada de *ciphertext stealing*[10], a qual dispensa a necessidade de se fazer o *padding*.

Com relação à capacidade de execução paralela de cifragem dos blocos, no CBC isso não é possível. Isso se deve ao fato de que para cifrar um bloco, primeiro é preciso aguardar o término da cifragem do bloco anterior a ele. Já com relação a decifragem, para decifrar um bloco apenas é necessário o bloco cifrado atual e o anterior. Como ambos estão prontamente disponíveis no início do processo, é possível decifrá-los em paralelo.

2.3.3 Modo *Cipher Feedback* (CFB)

Os modos CFB, OFB e CTR possuem como principal característica poderem operar com tamanhos de dados que não precisam ser múltiplos do tamanho do bloco da cifra. Também são caracterizados por utilizarem somente a função de cifragem da cifra de bloco em ambas as operações de cifragem e decifragem. Assim como o modo CBC, o modo CFB também possui uma espécie de encadeamento entre os blocos a serem cifrados e precisa utilizar um IV para gerar o bloco inicial.

No modo CFB, é preciso definir um tamanho de segmento específico, denotado pela letra s, em bits. Esse segmento representa uma parte do bloco a ser trabalhada e vai corresponder ao tamanho dos blocos de texto cifrado que serão gerados. Inicialmente é aplicado o processo de cifragem sobre o IV. O tamanho do IV é igual ao tamanho do bloco da cifra utilizada para a cifragem. Esse tamanho é denotado pela letra b, também em bits. O primeiro segmento cifrado (C_1) de tamanho s é obtido através de uma operação de `XOR` do segmento em texto claro (P_1) com os s bits mais significativos (MSB[11]) do IV cifrado. O restante do IV cifrado, ou seja, os $b - s$ bits, são descartados. Para o próximo bloco, os bits do IV original são deslocados $b - s$

[8] Sigla do inglês *Initialization Vector*.

[9] Contração do inglês *number used only once*.

[10] Basicamente, a técnica de *ciphertext stealing* consiste em pegar uma parte do penúltimo bloco de texto cifrado para complementar o tamanho do último bloco de texto claro a ser cifrado, a fim de torná-lo múltiplo do tamanho do bloco. Assim, é possível manter o tamanho dos dados cifrados igual ao tamanho dos dados em texto claro. Mais detalhes sobre essa técnica podem ser vistos no próprio adendo em [22].

[11] Sigla do inglês *most significant* bits.

bits à esquerda e são concatenados a ele os s bits do bloco cifrado produzido anteriormente (C_1). Esse novo IV passa a atuar como uma espécie de registrador de deslocamento. Essa parte do IV original concatenada com o bloco cifrado C_1 é submetida novamente ao processo de cifragem. Os s bits mais significativos da saída dessa nova cifragem são utilizados para se fazer um XOR com o segundo bloco de texto claro (P_2) para produzir o texto cifrado (C_2). Os $b - s$ bits são descartados e a operação se repete para os demais blocos.

O processo de decifragem funciona com a mesma lógica do processo de cifragem. Inicia com a cifragem do IV, com posterior XOR dos s bit mais significativos com o primeiro bloco cifrado (C_1), para se obter o bloco de texto claro original (P_1). Para os próximos blocos é feito o deslocamento do IV e a concatenação do bloco cifrado anterior (C_1), seguido pela sua cifragem. Depois, aplica-se uma operação de XOR sobre os s bits mais significativos dessa cifragem com o bloco cifrado C_2. Assim, obtém-se novamente o bloco em texto claro P_2. O processo se repete para os demais blocos. Os processos de cifragem e decifragem podem ser vistos na Figura 2.7.

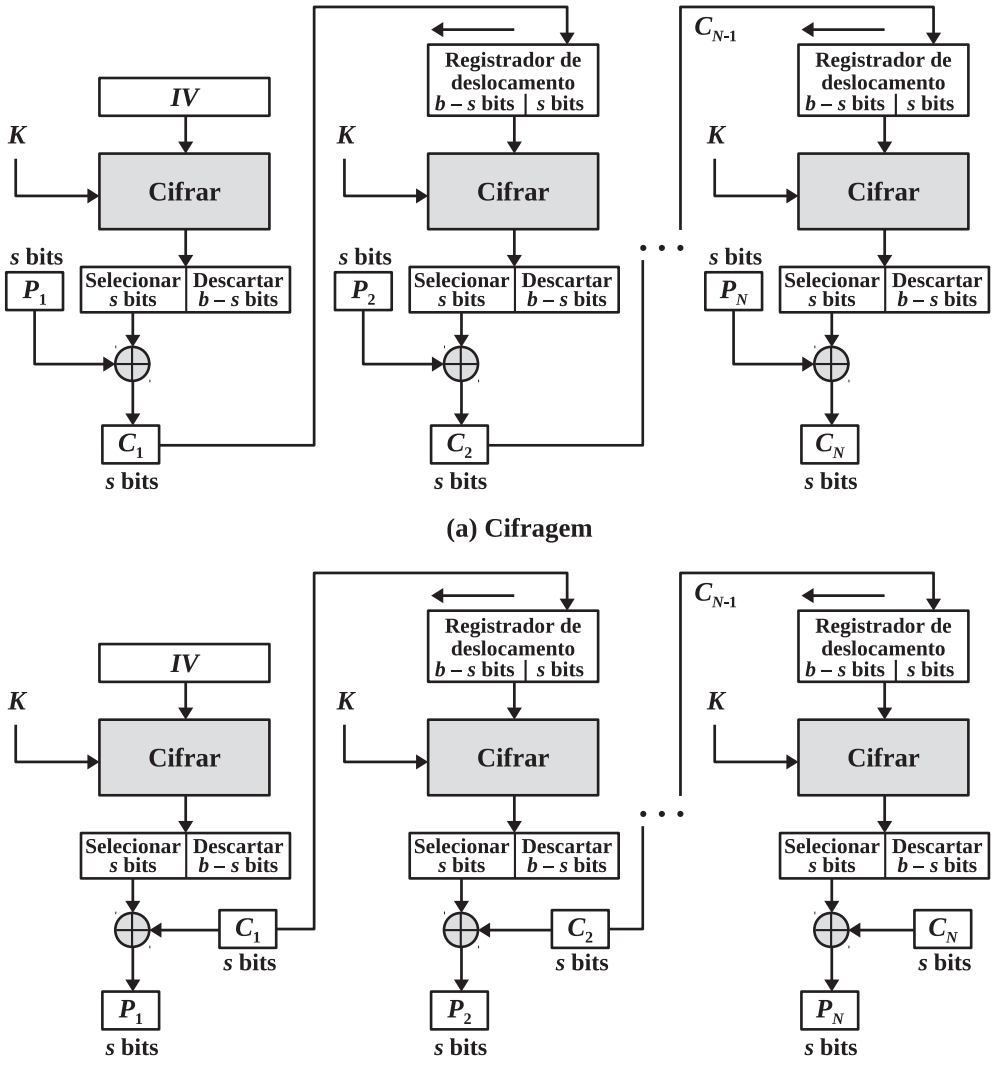


Figura 2.7: Modo de operação CFB. Adaptado de [16].

Assim como o CBC, a cifragem do modo CFB não tem como ser paralelizada. Na decifragem, para que seja possível paralelizá-la, primeiro é preciso gerar, a partir do IV inicial,

os demais blocos iniciais de forma sequencial. Após isso, a execução das etapas de cifragem[12] e da operação de XOR podem ser feitas de forma paralela.

2.3.4 Modo *Output Feedback* (OFB)

Comparado aos modos CBC e CFB, o modo OFB possui um funcionamento mais simplificado. Assim como o CFB e CTR, os processos de cifragem e decifragem de dados utilizam apenas a função de cifragem. No processo de cifragem, gera-se um *nonce* com tamanho igual ao do bloco da cifra sendo utilizada. Submete-se esse *nonce* ao processo de cifragem. O primeiro bloco cifrado C_1 é gerado simplesmente através da operação de XOR entre o *nonce* cifrado e o primeiro bloco de texto claro P_1. No segundo bloco, o *nonce* cifrado na etapa anterior é novamente submetido ao processo de cifragem e o resultado é submetido a um XOR com o bloco de texto claro P_2 para se obter o texto cifrado C_2. Esse procedimento se repete para os demais blocos.

Com relação ao último bloco, esse pode ter um tamanho menor do que o tamanho do bloco da cifra. Nesse caso, a operação é a mesma realizada para os demais blocos. A única diferença é que a etapa de XOR envolverá somente o bloco de texto claro P_N e os bits mais significativos do último bloco cifrado. Os demais bits são descartados.

O processo de decifragem é exatamente igual, mudando apenas as entradas para a etapa de XOR, que passam a ser os blocos cifrados. Ambas as operações podem ser vistas na Figura 2.8.

Ao contrário do modo CBC e CFB, no modo OFB é feito o uso de um *nonce*. Portanto, é imprescindível que ele nunca seja utilizado mais do que uma vez com a mesma chave. Caso isso não seja feito, os blocos cifrados com o mesmo par de *nonce* e chave podem ter sua confidencialidade comprometida. O *nonce* não precisa ser confidencial e, no caso específico do OFB, não precisa ser imprevisível [20].

Tanto o modo de cifragem quanto o modo de decifragem do OFB não podem ser paralelizados devido à dependência do bloco cifrado anteriormente. Porém, levando-se em consideração que para gerar os blocos cifrados é preciso inicialmente somente o *nonce*, é possível realizar sua cifragem antecipadamente. Depois, quando necessários, eles podem ser utilizados na etapa final de XOR para se obter efetivamente os dados cifrados ou decifrados.

2.3.5 Modo *Counter* (CTR)

O funcionamento do modo CTR é simples. Ele consiste na utilização de um contador o qual é incrementado a cada novo bloco processado. Esse contador é submetido ao processo de cifragem para depois ser realizado um XOR com o texto a ser cifrado ou decifrado. O resultado dessa etapa de XOR corresponde ao texto final cifrado ou decifrado. Os processos de cifragem e decifragem são idênticos, alterando-se apenas as entradas na etapa de XOR entre texto claro e cifrado. Ambos os processos podem ser vistos na Figura 2.9.

Esse modo também permite processar tamanhos de dados que não sejam múltiplos do tamanho do bloco da cifra utilizada. Seu funcionamento é igual ao do modo OFB. Quando há um último bloco com tamanho menor, faz-se um XOR somente com os bits mais significativos do contador cifrado. Os demais bits são descartados.

Apesar de sua simplicidade, o modo CTR tem uma parte bastante sensível: a construção do contador. Todos os contadores podem ser utilizados apenas uma única vez com a mesma

[12] Conforme mencionado anteriormente, no modo CFB somente é utilizada a função de cifragem, mesmo no modo de decifragem.

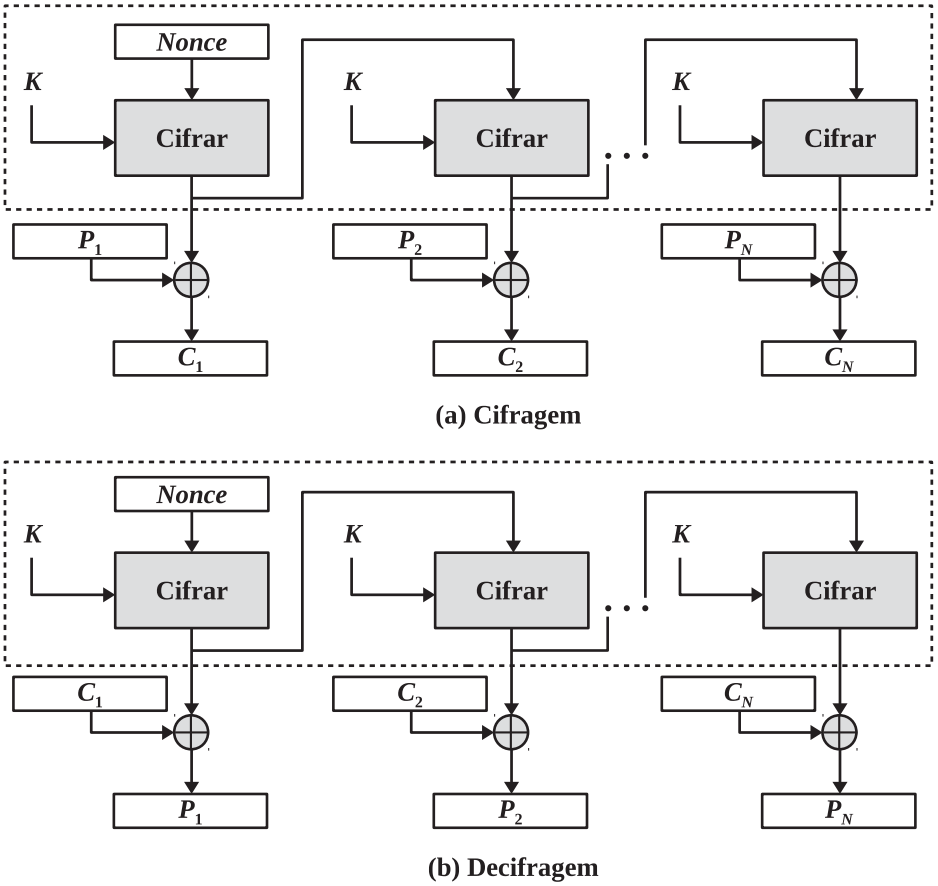


Figura 2.8: Modo de operação OFB. Adaptado de [16].

chave. Se essa regra não for obedecida, os dados cifrados usando os contadores repetidos com a mesma chave tem sua confidencialidade comprometida [20].

Também em [20], há duas sugestões de como esse contador pode ser construído. A primeira maneira é utilizar todos os bits do bloco a ser cifrado para armazenar o contador. Inicializa-se o contador em zero e simplesmente vai-se incrementado ele a cada novo bloco cifrado. Por exemplo, ao se utilizar o AES como cifra, o tamanho do bloco seria de 128 bits. Isso permitiria, com uma única chave, cifrar até 2^{128} blocos. O complicador desse método é ter que controlar o valor do contador utilizado no último processo de cifragem com uma determinada chave. Isso é essencial para que se possa incrementá-lo num próximo processo de cifragem que utilize a mesma chave.

A segunda maneira ajuda a evitar a necessidade de controlar o valor do contador entre processos de cifragem diferentes. Nesse caso, reserva-se uma quantidade de bits do bloco para guardar um *nonce* e os demais bits são utilizados para o contador. Novamente, usando como exemplo o AES, pode-se utilizar os 64 bits iniciais para armazenar o *nonce* e os 64 bits finais o contador. Isso ainda permitiria cifrar 2^{64} blocos de dados. Conforme anteriormente descrito, deve-se garantir que o *nonce* nunca seja utilizado mais do que uma vez com a mesma chave.

Apesar do cuidado necessário na construção do contador, o modo CTR tem vantagens significativas, conforme apontado por [17]: (i) Por não haver nenhuma dependência entre os blocos que são processados, tanto o processo de cifragem como o de decifragem podem ser totalmente paralelizados; (ii) O fato da cifragem ocorrer somente em cima do contador, e não do texto claro, permite que o processo de cifragem possa ser feito de forma antecipada. Assim, os dados cifrados são utilizados somente quando forem necessários na etapa de `XOR` com o

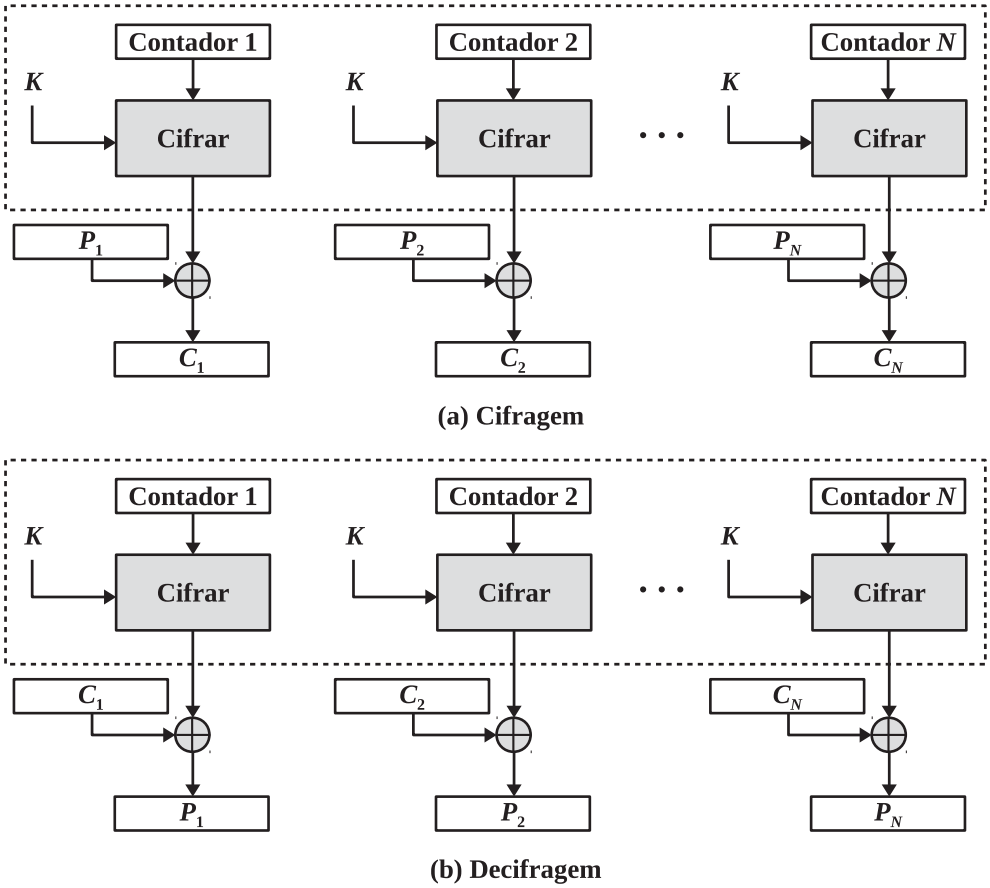


Figura 2.9: Modo de operação CTR. Adaptado de [16].

texto claro ou cifrado; (iii) No caso, por exemplo, da cifragem de um arquivo, o modo CTR permite realizar um acesso aleatório a qualquer parte do arquivo sem precisar realizar a cifragem ou decifragem das outras partes; (iv) Pelo fato dos processos de cifragem e decifragem serem idênticos, torna mais simples a sua implementação.

2.4 Demais cifradores simétricos e modos de operação

Além do AES, existem vários algoritmos de cifragem simétrica de blocos. Os parágrafos seguintes descrevem brevemente alguns exemplos de algoritmos que utilizam três esquemas diferentes de funcionamento: os baseados em rede de substituição-permutação (SPN[13]), rede de Feistel e no esquema Lai-Massey [23][24].

O AES faz parte da categoria de cifradores de blocos cuja estrutura é denominada rede de substituição-permutação. Cifradores dessa categoria tem em comum o fato de utilizarem rodadas de cifragem onde são aplicadas funções, chamadas S-Boxes e P-Boxes, as quais executam operações de substituição e permutação, além da adição da chave da rodada. Outros dois exemplos de cifradores dessa categoria são o Serpent e o Kuznyechik.

Serpent [25] foi outro algoritmo finalista na competição promovida pelo NIST na qual o AES foi o vencedor. Cifra blocos de 128 bits, com chaves de 128, 192 e 256 bits. Trabalha com quatro blocos de 32 bits e utiliza 32 rodadas de cifragem. O Kuznyechik [26] está definido

[13]Sigla do inglês *Substitution-Permutation Network*.

no padrão GOST[14] R 34.12-2015 e na RFC 7801. Trabalha com blocos de 128 bits, chaves de 256 bits e utiliza 10 rodadas de cifragem.

Outra categoria de cifradores de bloco refere-se àqueles cuja estrutura é baseada no que é denominado rede de Feistel. Uma rodada de cifradores desse tipo consiste na divisão do bloco a ser cifrado em duas partes. Um das partes sofre a ação de uma função específica de cifragem na qual também é utilizada como entrada uma chave da rodada. O resultado é adicionado a outra parte e após isso, as duas partes trocam de posição. Uma característica das funções de cifragem da rede de Feistel é que a mesma função é utilizada também no processo de decifragem. Dois exemplos de cifradores dessa categoria são o Twofish e Camellia.

Twofish [27] também foi um dos finalistas da competição do NIST. Tem relação com algoritmos de uma geração anterior a ele, Blowfish, e posterior, Threefish. Utiliza tamanhos de blocos e chaves iguais aos do AES, porém realiza 16 rodadas de cifragem. Tem como principal diferencial a utilização de S-Boxes variáveis, pois utiliza metade da chave de cifragem para gerá-las. O algoritmo Camellia [28] é recomendado pelo NESSIE[15] e pelo CRYPTEC[16], bem como pelo padrão ISO/IEC 18033-3:2010. Também usa blocos de 128 bits e chaves de 128, 192 e 256 bits. Para chaves de 128 bits, usa 18 rodadas de cifragem, para as demais, 24 rodadas. A cada seis rodadas utiliza uma função específica de transformação denominada função FL.

Também há uma categoria de cifradores de blocos denominada de Lai-Massey. Sua estrutura é semelhante à rede de Feistel. O bloco a ser cifrado também é dividido em duas partes e calcula-se a diferença entre elas. Essa diferença é utilizada como entrada na função de cifragem da rodada, junto com a chave da rodada. A saída da função é adicionada às duas partes. Exemplos de algoritmos dessa categoria é o IDEA e o IDEA NXT.

O IDEA [29] foi criado inicialmente como uma alternativa ao DES. Porém, já exitem publicações que descrevem alguns ataques que quebram sua segurança. Existe uma nova versão do algoritmo denominada IDEA NXT [30]. Na sua versão original, o IDEA trabalha com blocos de 64 bits e chaves de 128 bits. São executadas oito rodadas e meia de cifragem. Já o IDEA NXT, anteriormente conhecido como FOX, permite trabalhar com blocos de 64 bits e chaves de 128 bits, ou blocos de 128 bits e chaves de 256 bits. Em ambos os casos são executadas 16 rodadas de cifragem.

Com relação aos modos de operação para cifradores de bloco, foram descritos somente os modos básicos que garantem a confidencialidade no processo de cifragem. Existem outras SPs do NIST que descrevem modos adicionais.

Há a SP 800-38B [31] que descreve o funcionamento do modo CMAC. O modo CMAC utiliza os recursos de algoritmos de cifragem simétrica de blocos para gerar uma espécie de *hash* que permite verificar a autenticidade e integridade de dados. Outro modo está descrito na SP 800-38D [32]. Ela descreve o funcionamento do modo *Galois/Counter* (GCM), o qual provê, além da confidencialidade dos dados, sua autenticidade e integridade. Também há a SP 800-38E [33], que aprova a utilização do modo XTS usando AES, especialmente para prover confidencialidade ao armazenamento de dados em dispositivos que operam sobre blocos.

[14] GOST é um conjunto de padrões técnicos mantido por um conselho euro-asiático de padronização, metrologia e certificação (`http://www.easc.org.by/english/mgs_org_en.php`).

[15] Sigla do inglês *New European Schemes for Signatures, Integrity and Encryption*. Foi um projeto europeu em atividade entre 2000 e 2003 que identificou e avaliou a qualidade de desenvolvimento de vários algoritmos criptográficos. Teve objetivos parecidos com o processo de escolha do AES promovido pelo NIST.

[16] Um comitê para pesquisa e avaliação relacionados à criptografia, também com objetivos parecidos aos do NIST e NESSIE, porém mantido pelo governo japonês.

2.5 Considerações finais

Como parte da fundamentação teórica, neste capítulo foram apresentados assuntos relacionados à cifragem simétrica de blocos e os modos de operação. Foram vistos os principais conceitos envolvidos na cifragem simétrica de dados (Seção 2.1). Por ser o algoritmo no qual a biblioteca WAESlib está baseada, foram apresentadas algumas características do algoritmo de cifragem simétrica de blocos denominado AES (Seção 2.2).

Também foram descritos os principais modos de operação que podem ser utilizados com as cifras de bloco para se conseguir cifrar e decifrar de forma segura dados de tamanhos variados. Foram abordados os modos ECB, CBC, CFB, OFB e CTR (Seção 2.3). Foi dado maior ênfase ao modo CTR devido as suas características que permitem ser exploradas no contexto do processamento paralelo em GPU. Entre essas características estão a capacidade de ser totalmente paralelizado e permitir realizar a cifragem de dados de forma antecipada a sua utilização (Subseção 2.3.5).

Para complementar e finalizar o capítulo, foram brevemente apresentados outros exemplos de cifradores simétricos de blocos como o Serpent, Kuznyechik, Twofish, Camellia, IDEA e IDEA NXT. Também foram apresentados alguns modos de operação alternativos como o CMAC e o GCM (Seção 2.4).

Uma das aplicações básicas da cifragem de dados é sua utilização para tornar confidenciais os arquivos que são armazenados em disco. Sistemas de armazenamento criptográficos tem justamente essa função. Seu funcionamento está baseado no uso da cifragem simétrica e dos modos de operação vistos neste capítulo. O capítulo seguinte é dedicado especificamente para abordar alguns desses sistemas.

Capítulo 3

Sistemas de armazenamento criptográficos

Uma das funções básicas de um sistema operacional (SO) é oferecer uma camada de abstração clara e simplificada que permita às aplicações ter acesso a variados recursos de hardware como memória, processadores, dispositivos de armazenamento de dados, entre outros. Também se encarrega de gerenciar e controlar o acesso a esses recursos. No SO Linux essas funções estão implementadas em três componentes principais: componente de controle de entrada e saída (E/S), componente de gerenciamento de memória e componente de gerenciamento de processos. Eles formam o que é chamado de *kernel* do SO, estando situados em um espaço denominado espaço do *kernel*. [34][35][36].

Para entender como os sistemas de arquivos executam a cifragem de dados antes de armazená-los, é importante entender os subcomponentes do SO envolvidos no processo de armazenamento de dados. Esses subcomponentes fazem parte do componente de controle de E/S. No Linux, eles estão organizados em camadas e do nível mais baixo até o mais alto são os seguintes: driver de dispositivo, camada genérica de blocos/escalonador[1], sistema de arquivos e *Virtual File System* (VFS). Esses subcomponentes estão ilustrados na Figura 3.1. Na sequência, uma breve descrição de suas funções é apresentada.

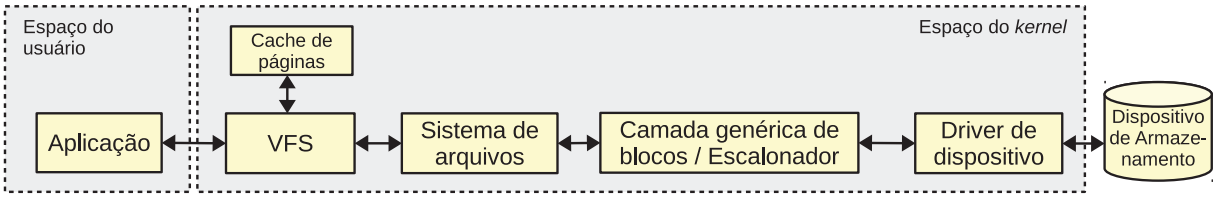


Figura 3.1: Subcomponentes do componente de E/S do SO Linux.

Na camada mais inferior está o driver de dispositivo. É através dele que se tem acesso e controle sobre o dispositivo de armazenamento onde os dados são efetivamente gravados. Antes das requisições de leitura e gravação chegarem ao driver de dispositivo, primeiro elas passam pela camada genérica de blocos, sofrendo a ação do escalonador de requisições. O escalonador utiliza algoritmos específicos que procuram ordenar e fundir requisições visando otimizar o acesso físico aos dispositivos de armazenamento.

Acima da camada genérica de blocos encontra-se a camada de sistema de arquivos. Sistemas de arquivos que são executados no espaço do *kernel* situam-se nela. Ela é a intermediária entre o VFS e camada genérica de blocos. Diferentes sistemas de arquivos possuem formas distintas de funcionamento. Por isso, é nessa camada onde estão implementas as várias funções associadas a arquivos e diretórios, as quais são chamadas através do VFS.

[1]Geralmente descrito na literatura simplesmente pelo termo em inglês *Block I/O* e *I/O Scheduler*.

Considerando que o Linux suporta uma quantidade considerável de diferentes sistemas de arquivos, é importante que haja uma forma padronizada de acessar as funções que operam sobre arquivos e diretórios. Essa é uma das principais funções do VFS. Ele funciona como uma camada de abstração onde estão definidas interfaces e estruturas de dados que são suportados pelos diferentes sistemas de arquivos. Dessa forma, para o VFS e o restante do *kernel*, há um conjunto de funções padronizadas que podem ser utilizadas para se trabalhar com arquivos e diretórios, independente do sistema de arquivos que estiver sendo utilizado.

Anexo ao VFS funciona o mecanismo de *cache* de páginas. Ele utiliza a memória RAM[2] para realizar *cache* de dados que são lidos e gravados através do VFS. Sua principal função é agilizar a leitura e gravação, reduzindo e otimizando a necessidade de acesso ao dispositivo onde efetivamente os dados estão armazenados.

Na camada mais superior está a aplicação. Porém, ainda há uma barreira existente entre a aplicação e o VFS. Essa barreira representa a divisão que existe entre dois espaços e modos distintos. O *kernel* do SO e seus componentes ocupam um espaço protegido, chamado espaço do *kernel*, sendo executados em um modo denominado modo do *kernel*. Uma aplicação normal ocupa um espaço chamado espaço do usuário, sendo executada no modo denominado modo do usuário.

Por questões de segurança e para garantir a integridade do SO, uma aplicação não pode acessar diretamente o espaço do *kernel*. Esse acesso só é possível através de uma interface específica denominada interface de chamadas de sistema. Ela é responsável por tratar uma chamada de sistema feita por uma aplicação e encaminhá-la a outro componente do *kernel* que a tratará. Também controla a alteração entre os modos de execução de usuário para *kernel* e vice-versa. As chamadas de sistema relacionadas a sistemas de arquivos estão definidas no VFS e são encaminhadas a ele.

A evolução dos SOs trouxe cada vez mais aperfeiçoamentos aos subcomponentes do sistema de entrada e saída do *kernel*. Os sistemas de arquivos e demais sistemas de armazenamento também acompanharam essa evolução. O aumento da flexibilidade de como eles se ligam aos subcomponentes de entrada e saída ajudaram na criação de diversos sistemas com recursos variados. Entre eles os que permitem cifrar os dados ao serem armazenados. São os sistemas de armazenamento criptográficos.

O fato da segurança da informação ter se tornado tão importante se reflete na atual quantidade existente de sistemas de armazenamento criptográficos. Em [37] pode-se verificar uma relação com mais de 60 sistemas. No mínimo, tais sistemas preocupam-se em atender a pelo menos uma das propriedades da segurança da informação: a confidencialidade. Há sistemas mais avançados que atendem a propriedades adicionais como integridade, disponibilidade, autenticidade, entre outras.

Considerando as soluções baseadas em software, há sistemas que apenas oferecem um recurso opcional de cifrar os dados do arquivo ao gravá-lo. Ou são ferramentas que permitem cifrar um ou mais arquivos de forma explícita através da sua utilização pelo usuário.

Outros, operam de forma mais transparente, cifrando arquivos e diretórios inteiros. Geralmente esses sistemas dependem de um sistema de arquivos sobre o qual funcionam, sendo chamados neste trabalho de sistema de arquivos de base (SAB), onde são efetivamente armazenados os arquivos e diretórios cifrados. Apesar de a maioria depender das funções de outro sistema de arquivos para armazenar os dados, eles são chamados de sistemas de arquivos criptográficos.

[2] Sigla do inglês *Random-Access Memory*.

Também há sistemas que operam num nível mais baixo, não exercendo nenhuma função de sistema de arquivos, apenas cifrando blocos de dados. Nesse caso, para se trabalhar com arquivos e diretórios, acima deles é necessário haver um sistema de arquivos em operação.

Este capítulo descreve alguns desses sistemas. Optou-se por abordar somente sistemas baseados em software e de código fonte aberto. Essas duas características são essenciais para um estudo mais aprofundado e consequente adaptação para que eles possam executar o processamento das funções de cifragem em GPUs.

A Seção 3.1 apresenta as aplicações com recursos de criptografia. A Seção 3.2 apresenta os sistemas de arquivos criptográficos baseados no *File System in User Space* (FUSE). A Seção 3.3 apresenta os sistemas de arquivos baseados no *Network File System* (NFS). A Seção 3.4 apresenta os sistemas de arquivos criptográficos executados no espaço do *kernel*. Na Seção 3.5 são apresentados os sistemas de armazenamento criptográficos que operam sobre dispositivos de blocos. Por último, na Seção 3.6, comenta-se brevemente sobre outros tipos de sistemas.

3.1 Aplicações com recursos de criptografia

As aplicações com recursos de criptografia podem ser de dois tipos: as que são utilizadas para realizar operações criptográficas sobre arquivos ou mensagens e as que apenas oferecem a opção de salvar um arquivo de forma cifrada. São aplicações que rodam no espaço do usuário, onde também as funções de cifragem sobre os dados são aplicadas. A Figura 3.2[3] ilustra sua localização (quadro laranja) no espaço do usuário e onde efetivamente são executadas as funções de cifragem.

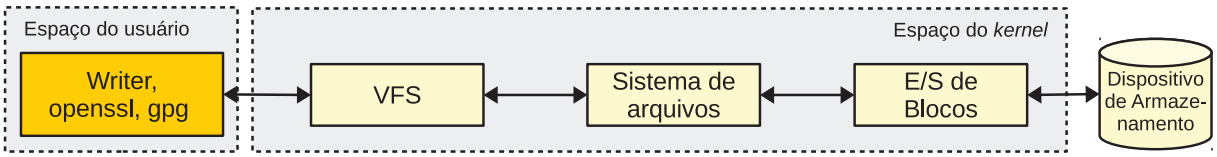


Figura 3.2: Localização no espaço do usuário das aplicações com recursos de criptografia.

Exemplos do primeiro caso são ferramentas como o `openssl` [38] e o `gpg` [39]. O `openssl` pode ser utilizado diretamente na linha de comando para cifrar um ou mais arquivos fazendo uso de diversos algoritmos de cifragem e modos de operação. Implementa uma biblioteca com recursos de criptografia que pode ser utilizada no desenvolvimento de outras aplicações. O `gpg` é uma implementação livre e completa do padrão OpenPGP. Oferece ferramentas que também podem ser utilizadas na linha de comando ou integradas em outras aplicações. Possui recursos avançados de criptografia assimétrica, podendo fazer a cifragem e assinatura de dados e mensagens. Apresenta recursos avançados de gerenciamento de chaves.

Exemplos do segundo caso são os aplicativos que fazem parte de suítes de escritório como o Writer do LibreOffice [40]. Geralmente esses aplicativos oferecem o recurso de salvar um arquivo utilizando uma senha. A senha fornecida é utilizada num processo de derivação de chave. A chave resultante é utilizada para cifrar o conteúdo do arquivo. Sempre que o arquivo é aberto, a senha precisa ser fornecida. O LibreOffice, desde a versão 3.5, utilizada como algoritmo de cifragem o AES e o modo de operação CBC.

[3] Para facilitar a visualização, nessa ilustração e nas dos demais sistemas, a camada genérica de blocos/escalonador é apresentada apenas como "E/S de blocos". Foi omitida a camada de driver de dispositivo e o subcomponente correspondente ao mecanismo de *cache* de páginas.

Esses dois tipos de aplicações, apesar da vantagem de serem simples e fáceis, possuem desvantagens significativas: necessitam da atribuição e fornecimento de uma senha sempre que um arquivo cifrado é criado e aberto; cópias temporárias de arquivos não são cifradas; não é garantida a compatibilidade entre aplicações diferentes; os nomes dos arquivos e metadados não são cifrados. Essas desvantagens resultam em um nível baixo de transparência, pois necessitam de um nível de interação alta com o usuário. Dentre os sistemas descritos neste capítulo, possuem o pior desempenho.

3.2 Sistemas de arquivos criptográficos baseados no FUSE

Uma das desvantagens mais significativas no processo de cifragem dos arquivos descrita na seção anterior é a baixa transparência associada aos processos de cifragem de arquivos. Uma das soluções encontradas para contornar esse problema foi a criação de uma forma de extensão para sistemas de arquivos já existentes. Assim, além das funções normais de um sistema de arquivos, eles também podem realizar operações de cifragem sobre os dados quando esses são lidos ou gravados.

Um dos caminhos adotados para desenvolver tais extensões foi a utilização dos recursos oferecidos por um módulo do *kernel* do Linux chamado FUSE [41]. Ele consiste em uma parte executada no espaço do *kernel*[4] e outra no espaço do usuário[5]. As funções adicionais oferecidas por sistemas desse tipo são implementadas no *daemon*. Essa característica permite criar sistemas de arquivos que rodam no espaço do usuário, simplificando seu desenvolvimento. Geralmente eles dependem de outro SAB onde os dados cifrados são efetivamente gravados.

O EncFS [42] é um exemplo de sistema baseado no FUSE. Para entender melhor seu funcionamento é interessante descrever o que acontece quando uma aplicação faz uma chamada de sistema do tipo `read()` sobre um arquivo que se encontra num sistema de arquivos EncFS. Esse processo está ilustrado na Figura 3.3 e descrito na sequência. As setas em verde indicam o caminho de ida da requisição. As setas em azul indicam o caminho de volta, onde os dados lidos são devolvidos à aplicação.

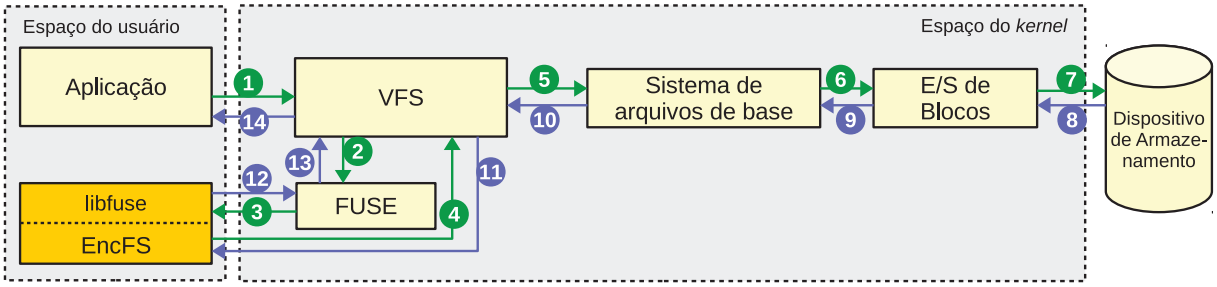


Figura 3.3: Caminho percorrido pela requisição e pelos dados numa operação de leitura do EncFS.

Inicialmente a chamada de sistema é encaminhada ao VFS (1). O VFS a repassa ao módulo FUSE (2). O módulo FUSE se comunica com o EncFS através da libfuse (3). No *daemon* EncFS há uma função implementada para tratar a chamada `read()`. Essa função se encarrega de fazer outra chamada ao VFS (4) para ler os dados cifrados do sistema de arquivos

[4]A parte que executa no espaço do *kernel* é o módulo `fuse.ko`. O módulo FUSE está incluído e é mantido nos repositórios oficiais do *kernel* do Linux.

[5]Geralmente desenvolve-se um componente de software que fica rodando como um *daemon* e no qual é utilizada a biblioteca libfuse.

onde eles estão efetivamente armazenados. Dessa vez, o VFS repassa a requisição ao SAB (5). O sistema de arquivos repassa à camada de E/S de blocos (6) que prepará a requisição de leitura para o dispositivo físico onde os dados estão gravados (7). Após obter os dados, eles percorrem o caminho inverso até serem entregues à aplicação (8-14). É interessante observar que quando o VFS entrega os dados ao EncFS (11), eles ainda se encontram cifrados. Portanto, antes de serem encaminhados ao módulo FUSE (12), VFS (13) e aplicação (14), os dados são decifrados.

O EncFS realiza a cifragem dos dados utilizando o modo de operação CBC. Devido a isso, para que seja possível acessar aleatoriamente o conteúdo dos arquivos sem precisar cifrá-los ou decifrá-los completamente, a granularidade da cifragem dos dados contidos num arquivo acontece em blocos[6]. O tamanho desses blocos são definidos quando da criação do sistema de arquivos. Por padrão o EncFS cifra blocos de 1 KiB usando os cifradores AES ou Blowfish, disponíveis na biblioteca OpenSSL. Para cifrar nomes de arquivos e blocos com menos de 1 KiB é utilizado o modo CFB.

Quando um sistema de arquivos EncFS é criado ou montado, uma senha é solicitada. É utilizada uma função de derivação (PBKDF2[7]) para se obter uma chave a partir da senha informada. Essa chave derivada é utilizada para cifrar outra chave, a chave de volume, a qual efetivamente é utilizada para cifrar os nomes e dados dos arquivos. A chave de volume cifrada é armazenada num arquivo específico do SAB. Esse nível indireto de utilização de chaves permite trocar a senha principal de acesso ao sistema de arquivos sem precisar recifrar todos os dados.

Na cifragem de nomes de arquivos, o IV utilizado é derivado da aplicação de uma função de *hash* (HMAC[8]) sobre o nome do arquivo. Na configuração padrão é utilizado um IV diferente para cada arquivo. Esse IV é gerado aleatoriamente e armazenado no cabeçalho do arquivo. A cifragem dos blocos pertencentes ao arquivo utiliza um IV diferente para cada bloco. O IV de um bloco é obtido através da aplicação de uma função de *hash* (HMAC) que utilizada quatro parâmetros: chave de volume, IV de volume, IV de arquivo e número do bloco.

No EncFS, cada arquivo ou diretório corresponde a um arquivo ou diretório no SAB. Os metadados dos arquivos e diretórios (permissões, datas de acesso e alteração, identificação de proprietário e do grupo, etc) são os mesmos em ambos os sistemas de arquivos (EncFS e base). Portanto, os metadados não são cifrados. O mesmo se aplica à estrutura hierárquica dos diretórios.

Outra solução desenvolvida de sistema de arquivos baseada no FUSE é o CryFS [43][44]. Seu núcleo de funcionamento também está baseado em um *daemon* que roda no espaço do usuário. A comunicação com os demais componentes do sistema de armazenamento do Linux é feita da mesma forma que o EncFS. Sua estrutura interna está organizada em três camadas principais: *Blockstore*, *Blobstore* e *Filesystem*. Além de uma camada auxiliar chamada *fs++*.

A camada *Blockstore* é responsável por ler a gravar dados em blocos de tamanhos fixos. Esse tamanho é configurável e pode variar de 8 a 32 KiB. É nela que ocorrem as operações de criptografia. Cada bloco armazenado corresponde a um arquivo cifrado gravado no SAB. A

[6]Nesse caso, bloco se refere a um conjunto de dados sobre o qual o EncFS realiza operações de cifragem. É importante não confundir com o tamanho de bloco do SAB. Também não confundir com o tamanho de bloco de um dispositivo de blocos, o qual geralmente corresponde ao tamanho do setor de um disco.

[7]Sigla do inglês *Password-Based Key Derivation Function 2*. Essa técnica usa uma função pseudoaleatória para geração de *hashes* (HMAC-SHA1, HMAC-SHA2). As entradas para essa função são uma senha e um valor aleatório (denominado *salt*). A aplicação dessa função é repetida uma determinada quantidade de vezes e gera como resultado final uma chave que pode ser utilizada em operações de criptografia. Maiores detalhes podem ser vistos em https://www.rfc-editor.org/info/rfc8018.

[8]Sigla do inglês *keyed-Hash Message Authentication Code*. É uma técnica para geração de *hashes* utilizando uma função criptográfica de *hash* (MD5 ou SHA1) e uma chave criptográfica. Maiores detalhes podem ser vistos em: https://tools.ietf.org/html/rfc2104.

camada *Blobstore* é responsável por lidar com *blobs*[9] de tamanhos variáveis. Ela mapeia os *blobs* para serem armazenados em um ou mais blocos. A camada *Filesystem* é responsável por mapear arquivos e diretórios em *blobs*. Um arquivo ou diretório corresponde a um *blob*. No caso do CryFS para Linux, a camada *fs++* faz a interface com a libfuse.

Um arquivo armazenado no CryFS não necessariamente vai corresponder a um arquivo armazenado no SAB, como acontece no EncFS. No CryFS, um arquivo é mapeado para um *blob*, porém de acordo com seu tamanho, ele pode ser mapeado para um ou mais blocos. Esse mapeamento e o armazenamento em blocos cifrados de tamanhos iguais garante a confidencialidade dos metadados dos arquivos e da estrutura hierárquica dos diretórios. Esse é um diferencial em relação ao EncFS.

Os cifradores simétricos suportados são o AES, MARS, Twofish, Serpent e CAST disponíveis na biblioteca Crypto++. Os modos de operação são o GCM[10] e CFB. A utilização do modo de operação GCM garante também a integridade dos dados. É utilizado um IV para cada bloco, sendo o mesmo armazenado de forma não criptografada no início do bloco.

Comparados às aplicações com recursos de criptografia, sistemas de arquivos criptográficos baseados no FUSE adicionam um nível a mais de transparência ao permitirem cifrar todos os arquivos e subdiretórios armazenados neles. Também oferecem um grau razoável de controle ao permitir atribuir uma chave para cada diretório cifrado. Comparados aos sistemas de arquivos criptográficos que são executados no espaço do *kernel*, são mais fáceis de serem desenvolvidos.

Uma desvantagem significativa está associada ao seu desempenho: os dados precisam cruzar diversas vezes o espaço do usuário e do *kernel*. Isso acarreta uma sobrecarga devido à necessidade constante de trocas de contexto. Comparados aos sistemas de armazenamento criptográficos que operam sobre blocos, os quais permitem cifrar uma partição ou disco inteiros, os sistemas baseados no FUSE tem uma atuação mais restrita. Apesar de poder cifrar o conteúdo inteiro de um diretório, ao fazer o uso normal dos arquivos, o próprio SO ou as aplicações podem armazenar cópias desses arquivos ou partes deles em partições como o `/tmp` ou *swap*. Nesse caso, esses dados não serão cifrados.

3.3 Sistemas de arquivos criptográficos baseados no NFS

Outro caminho que alguns desenvolvedores optaram para desenvolver seus sistemas foi o de adaptar e aperfeiçoar os componentes que fazem parte do NFS[11].

O *Cryptographic File System* (CFS) [45] foi um dos primeiros sistemas de arquivos criptográficos desenvolvidos. A sua implementação seguiu um caminho menos complexo, sendo seu protótipo baseado numa versão modificada do *daemon* do NFS, o qual roda no espaço do usuário. Essa versão modificada, chamada cfsd, se comunica com o espaço do *kernel* através da interface do NFS. A parte cliente roda no espaço do *kernel* e se comunica com o servidor através de *Remote Procedure Calls* (RPCs).

A Figura 3.4 ilustra o fluxo de dados do CFS. A numeração das setas indicam a ordem e por quais componentes os dados passam numa operação de gravação. O quadro laranja indica

[9] Contração do inglês *binary large objects*.

[10] Sigla do inglês *Galois Counter Mode*. GCM é outro modo de operação para cifradores de bloco simétricos. Funciona como uma variação do modo CTR. Gera como saída, além do texto cifrado, um *tag* de autenticação que é utilizado para verificar a integridade e autenticidade dos dados cifrados. Mais detalhes podem ser vistos em [32].

[11] O NFS permite acessar remotamente arquivos armazenados e um sistema de arquivos hospedado em um servidor. Esse sistema de arquivos é exportado, podendo ser montado e acessado em outro computador como se fosse um sistema de arquivos local.

o servidor NFS modificado, o `cfsd`. Nesse cenário o servidor NFS está em execução na mesma máquina da aplicação, por isso a comunicação acontece através da interface de rede *loopback*.

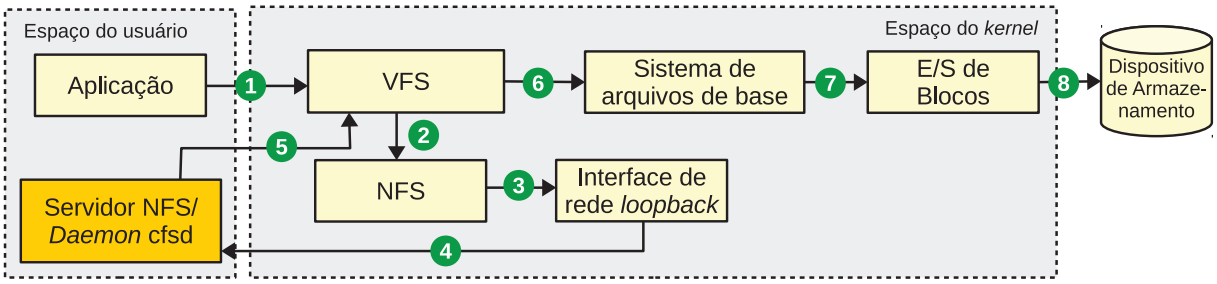


Figura 3.4: Fluxo de dados no CFS numa operação de gravação.

O CFS possui ferramentas específicas para criar e montar um diretório. Todos os dados de arquivos e diretórios armazenados nele serão cifrados, assim como seus nomes. Metadados não são cifrados. Para cifragem o CFS utiliza o DES e como modo de operação um esquema híbrido entre ECB e OFB. No modo de operação normal o CFS não utiliza IVs.

O *Transparent Cryptographic File System* (TCFS) [46] é outro exemplo de sistema desenvolvido com base no NFS. Porém seu desenvolvimento ficou concentrado na parte cliente do NFS, a qual roda no espaço do *kernel*. Portanto, todo processo de cifragem dos dados ocorre nela, para depois ser encaminhada a um servidor NFS normal. A comunicação entre a parte servidora e cliente é feita normalmente através do protocolo NFS. No quadro laranja da Figura 3.5 está ilustrada a parte modificada do NFS, correspondente ao TCFS, bem como sua localização no espaço do *kernel*.

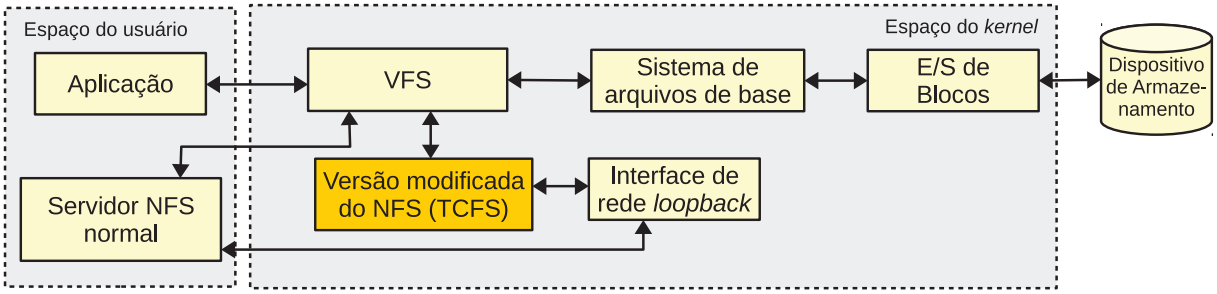


Figura 3.5: Localização no espaço do *kernel* do TCFS.

Um diferencial do TCFS é a possibilidade de acesso compartilhado a um arquivo cifrado por com um grupo de usuários. Ele utiliza um mecanismo denominado *Threshold Secret Sharing Scheme* para compartilhar uma chave criptográfica entre usuários de um determinado grupo. Esse esquema permite determinar uma quantidade mínima de usuários ativos no sistema de tal forma que, somente após atingida essa quantidade mínima, é possível reconstruir a chave completa e acessar o arquivo compartilhado pelo grupo. Para gerenciamento de usuários, senhas e grupos o TCFS possui ferramentas próprias.

O sistema de cifragem dos arquivos no TCFS é um pouco mais complexo do que no CFS. Todo arquivo possui um cabeçalho com algumas informações como versão do TCFS, número de identificação do cifrador de bloco utilizado e chave de cifragem do arquivo. Quando um arquivo é cifrado pelo TCFS pela primeira vez, uma chave aleatória é gerada e a mesma é cifrada com a chave-mestra do usuário. Essa chave cifrada é armazenada no cabeçalho do arquivo. Os arquivos são divididos em blocos. Cada bloco possui duas partes: uma parte de dados e uma etiqueta (*tag*) de autenticação. Cada bloco possui uma chave de cifragem diferente.

Essa chave é gerada através da aplicação de uma função de *hash* sobre a junção da chave do arquivo e o número do bloco. Cada bloco é cifrado com essa chave. Também é feito um *hash* sobre a junção dos dados do bloco e da chave do bloco. Esse *hash* é armazenado na etiqueta de autenticação. Com isto, além da propriedade de confidencialidade, o TCFS também garante a integridade dos arquivos cifrados.

Os cifradores de bloco utilizados pelo TCFS são implementados através de módulos do *kernel*, portanto ele pode utilizar qualquer cifrador que seja implementado dessa forma. O modo de operação utilizado é o CBC.

Comparado ao CFS, o TCFS apresenta recursos a mais como possibilidade de cifrar arquivos individualmente, compartilhar arquivos cifrados com um grupo de usuários, garantir a integridade dos arquivos cifrados, facilitar o processo de alteração de chaves sem a necessidade de recifrar todo o conteúdo dos arquivos.

Devido ao fato do CFS e TCFS usarem o modelo de funcionamento do NFS, eles necessitam constantemente copiar *buffers* entre a parte que roda no espaço do usuário e a parte que roda no espaço do *kernel*. Esse fato também ocasiona uma quantidade alta de trocas de contexto. Por isso, assim como os sistemas baseados no FUSE, sofrem um impacto significativo no seu desempenho. Ambos também não realizam a cifragem de metadados e estrutura hierárquica de diretórios.

3.4 Sistemas de arquivos criptográficos no espaço do *kernel*

Outra categoria de sistemas de arquivos criptográficos são aqueles cujos componentes que executam as operações de criptografia são executados no espaço do *kernel*, sendo desenvolvidos como módulos do *kernel*.

Quando ocorrem operações de leitura e escrita sobre arquivos ou diretórios armazenados em sistemas dessa categoria, elas são repassadas pelo VFS ao módulo correspondente do sistema de arquivos criptográfico que realiza a cifragem/decifragem dos dados. Após concluídas as operações criptográficas, o módulo se encarrega de fazer novas requisições ao VFS que as repassa ao SAB onde os dados são efetivamente lidos ou gravados.

Um exemplo de sistema de arquivos criptográfico dessa categoria é o NCryptfs [47]. Ele é o resultado do aperfeiçoamento do Cryptfs. O Cryptfs foi a primeira aplicação prática de uso da FiST[12] para a geração de um sistema de arquivos com recursos de criptografia no espaço do *kernel* [49]. Os principais recursos adicionados ao NCryptfs foram o suporte a múltiplos usuários, múltiplas chaves e um mecanismo adicional de autenticação e autorização. Com um controle próprio de grupos de usuários e o uso de algumas permissões a mais do que as normalmente presentes em sistemas Unix, o Ncryptfs permite aos usuários delegar acesso a um diretório cifrado. Inclusive permite controlar por quanto tempo esse acesso ficará ativo. A Figura 3.6 ilustra o posicionamento no *kernel* do NCryptfs.

Na sua implementação original, o NCryptfs usa Blowfish ou AES. Utiliza como modo de operação o CFB. Os dados são cifrados em blocos correspondentes ao tamanho da página do SO. Esse processo de cifragem dos blocos usa um IV em conjunto com a chave de cifragem. Ambos são submetidos a uma operação de `XOR` com o número do *inode* do arquivo e o número da página. Para o nome dos arquivos, o IV é submetido a um `XOR` com o número do *inode* do

[12] A FiST permite descrever sistemas de arquivos utilizando uma linguagem de alto nível. A partir de uma ferramenta específica (fistgen) é possível gerar código fonte de módulos do *kernel* para diferentes sistemas operacionais (Solaris, FreeBSD e Linux) [48].

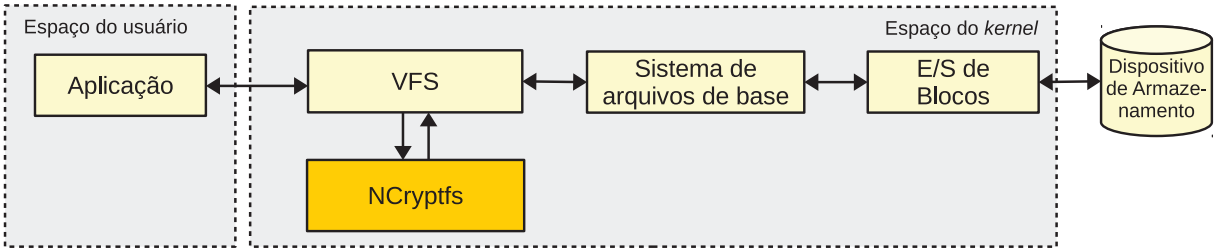


Figura 3.6: Localização no espaço do *kernel* do NCryptfs.

diretório. A senha fornecida, quando se monta um diretório cifrado, também passa pela função de derivação PBKDF2 para a geração da chave.

O eCryptfs [50] é outro exemplo de sistema de arquivos derivado do Cryptfs. É mantido e faz parte dos repositórios oficiais do Linux. Ao contrário dos outros sistemas apresentados, possui módulos adicionais como um módulo PAM (*Pluggable Authentication Module*), módulo PKI (*Public Key Infrastructure*) e um módulo TPM (*Trusted Platform Module*). O módulo PAM pode ser usado para capturar a senha do usuário e armazená-lo no chaveiro da sessão do usuário. Essa senha pode ser utilizada para gerar outra chave, a qual é utilizada para cifrar as chaves individuais de cifragem dos arquivos. O módulo PKI permite ter acesso ao chaveiro do usuário controlado pelo GnuPG. Usando a senha armazenada no chaveiro da sessão do usuário, através do módulo PAM, o módulo PKI pode decifrar as chaves privadas do usuário no chaveiro controlado pelo GnuPG, permitindo ao eCryptfs também fazer uso de cifragens assimétricas sobre chaves. O módulo TPM permite fazer uso de chaves privadas que ficam armazenadas em dispositivos TPM.

Seu posicionamento no espaço do *kernel* é igual ao do NCryptfs, como pode ser visto na Figura 3.7. O funcionamento dos módulos PAM, PKI e TPM dependem de algumas APIs que só estão acessíveis a partir do espaço do usuário. Por isso, o eCryptfs possui um *daemon* rodando nesse espaço que permitem ter acesso a essas APIs.

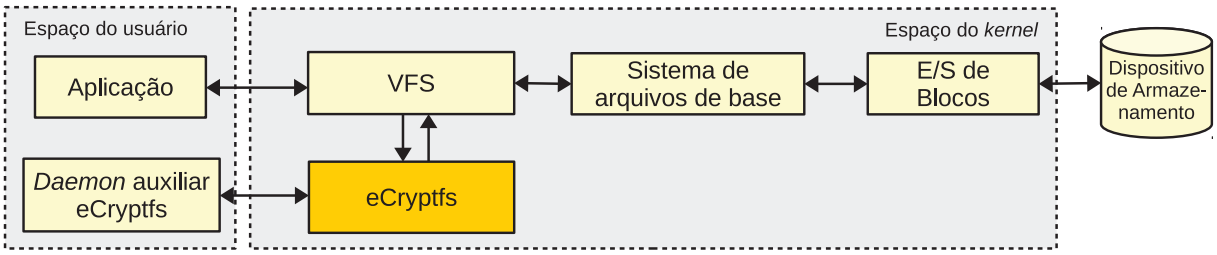


Figura 3.7: Posicionamento no *kernel* do eCryptfs e do seu *deamon* auxiliar.

No eCryptfs todos os metadados de cifragem[13] dos arquivos são armazenados com o arquivo, num cabeçalho. Quando um arquivo é criado, é gerada uma chave aleatória para cada arquivo. Dentro dos arquivos os dados cifrados são divididos em pedaços, os quais, assim como o NCryptfs, possuem o tamanho da página do SO. Esses pedaços são organizados em grupos e cada grupo de pedaços possui um conjunto de IVs. A cifragem desses pedaços é feita utilizando-se os IVs pertencentes ao grupo de pedaços e a chave gerada para o arquivo. Quando operando no modo que utiliza autenticação com chaves públicas, a chave do arquivo é cifrada com a chave pública do usuário e armazenada no cabeçalho do arquivo. Essa forma de funcionamento permite

[13]Metadados de cifragem referem-se a informações relacionadas às diversas operações criptográficas realizadas por um sistema criptográfico. Não estão relacionados aos metadados normais de um arquivo (permissões, identificação do usuário e grupo, datas de acesso e modificação, etc).

que o arquivo seja acessado por mais de um usuário, pois a chave de cifragem do arquivo se mantém a mesma, sendo guardado no cabeçalho do arquivo apenas uma versão cifrada dela para cada usuário que possuir acesso.

Para as operações de cifragem simétrica sobre os dados, o eCryptfs faz uso da API criptográfica do *kernel* do Linux (Crypto API). Isso permite utilizar diferentes cifradores de blocos que estejam disponíveis nela. O modo de operação é o CBC e, conforme descrito anteriormente, por cifrar os dados separadamente em pedaços, permite realizar o acesso aleatório a partes dos arquivos de forma eficiente. Assim como o NCryptfs, informações de metadados dos arquivos não são cifradas.

No geral, as soluções de sistemas de arquivos criptográficos que são executados no espaço do *kernel* possuem a desvantagem de não cifrar as informações de metadados dos arquivos e ser possível identificar a estrutura hierárquica dos diretórios. Arquivos temporários e armazenados em páginas de memória que sofreram *swap* não são cifrados.

Comparando-se com os sistemas descritos nas seções anteriores, as vantagens são caracterizadas por um nível maior de transparência no processo de cifragem, bem como de controle na escolha de quais arquivos e diretórios serão cifrados. Por serem sistemas que rodam no espaço do *kernel*, requerem uma quantidade menor de trocas de contexto e cópias de dados entre o espaço do usuário e do *kernel*. Consequentemente, seu desempenho é significativamente superior aos sistemas que rodam no espaço do usuário como aqueles baseados no FUSE.

3.5 Sistemas de armazenamento criptográficos que operam sobre dispositivos de blocos

São sistemas situados abaixo da camada de software na qual operam os sistemas de arquivos normais e acima da camada de software correspondente à camada de E/S de blocos. Em vez de operarem cifrando o conteúdo de arquivos, cifram os blocos de dados advindos dos sistemas de arquivos antes de serem efetivamente armazenados em dispositivos de blocos, como por exemplo, os discos. Sistemas deste tipo cifram os dados de uma forma mais abrangente, podendo cifrar discos inteiros, partições, volumes lógicos e volumes do tipo *Redundant Array of Independent Disks* (RAID).

Alguns exemplos de sistemas que operam neste nível são o dm-crypt do Linux e o sistema multiplataforma VeraCrypt. A Figura 3.8 ilustra seu posicionamento no espaço do *kernel* Linux, bem como sua ligação à camada de E/S de blocos através do mapeador de dispositivo[14].

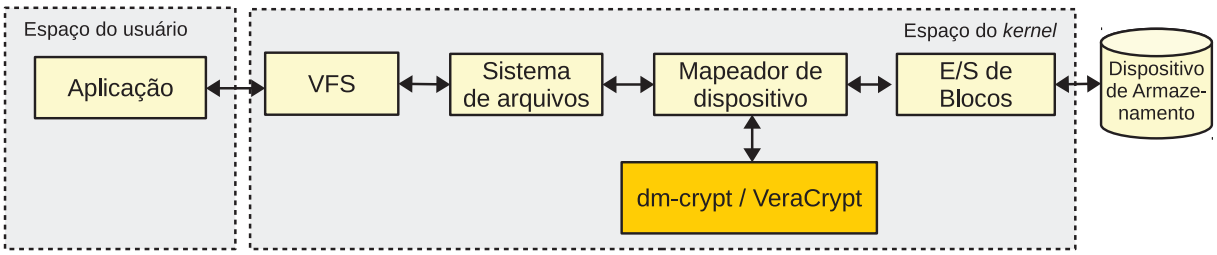


Figura 3.8: Localização no espaço do *kernel* do dm-crypt e VeraCrypt.

O dm-crypt [51] também está incluso nos repositórios oficiais do *kernel* do Linux. Ele funciona utilizando o mapeador de dispositivo, uma camada do sistema de E/S que permite criar dispositivos virtuais de blocos, os quais interagem com a camada de E/S de blocos. Dessa forma,

[14]Tradução do termo em inglês "*device-mapper*".

assim como acontece com o VFS, é possível adicionar novas funcionalidades a esses dispositivos. Utilizando os recursos do mapeador de dispositivo para se acoplar a camada de E/S de blocos, o dm-crypt consegue adicionar de forma transparente recursos de cifragem sobre blocos de dados antes deles serem efetivamente lidos ou gravados em um dispositivo de blocos.

Quando um usuário cria um dispositivo dm-crypt, ele pode escolher os algoritmos de cifragem e os modos de operação a serem utilizados entre aqueles presentes na Cripto API do Linux. Também define a chave a ser utilizada no processo de cifragem e a forma de geração dos IVs. A geração dos IVs é baseada nos números dos setores. Após criado, o dispositivo aparece no diretório `/dev` e pode ser utilizado como um disco normal, podendo ser particionado, utilizado diretamente para se criar um sistema de arquivos e até usado em arranjos de *Logical Volume Manager* (LVM). Todos os dados lidos e gravados nesse dispositivo são cifrados de forma transparente.

As informações de criptografia associadas a um dispositivo dm-crypt são armazenadas no formato *Linux Unified Key Setup* (LUKS) [52][53]. LUKS é uma forma padronizada de armazenar metadados de criptografia no cabeçalho de partições, visando simplificar a configuração e o gerenciamento dos dispositivos cifrados. A Figura 3.9 ilustra o formato do cabeçalho LUKS.

Figura 3.9: Formato do cabeçalho de uma partição LUKS. Adaptado de [53].

Esse cabeçalho começa com o LUKS phdr (LUKS *partition header*). No phdr estão contidas informações como qual cifrador foi utilizado, modo de operação, tamanho da chave utilizada, número de identificação da partição e um *checksum* da chave-mestra utilizada para cifrar o dispositivo. Na sequência estão os *slots* onde são armazenados os denominados *key material* (KM). Em cada *slot* é armazenada uma cópia cifrada da chave-mestra utilizada para cifrar o dispositivo e mais algumas informações utilizadas pelas funções de derivação de chave. Existem até oito *slots*. Isso permite que sejam utilizadas até oito chaves diferentes para acessar o dispositivo cifrado. O padrão LUKS também se encarrega de cuidar de outros detalhes importantes de segurança, utilizando funções de derivação de chave (PBKDF2) e mecanismos antiforenses (AF-Splitter[15]) para dificultar a recuperação de chaves que foram removidas [53].

Com relação ao VeraCrypt, seu desenvolvimento iniciou por volta de 2013 e continua sendo mantido. É resultado de um *fork* do TrueCrypt, outro sistema mais antigo cujo desenvolvimento iniciou em 2004, porém foi descontinuado em 2014. É um sistema de código fonte aberto, multiplataforma, podendo ser executado no Windows, Mac OS e Linux. Ele pode ser utilizado para criar volumes cifrados que ficam contidos num único arquivo, volumes que ocupam uma partição ou disco inteiro e até mesmo partições ou discos de sistema.

Outra característica interessante é a capacidade de criar volumes cifrados escondidos, os quais são armazenados dentro de um outro volume cifrado. Os dados armazenados nesse volume escondido utilizam a área de espaço livre do volume cifrado no qual ele está contido [54][55].

Os metadados de cifragem são armazenados no cabeçalho de cada volume criado. São 128 KiB de informações. Uma cópia de segurança também é armazenada no final do volume. Os detalhes das informações contidas nesse cabeçalho podem ser vistas em [56]. Essas informações

[15] A técnica de AF-Splitter consiste em pegar uma chave e inflar seu tamanho até um valor desejado. Isto permite fazer com que ela ocupe uma quantidade maior de blocos ao ser armazenada em disco. Caso a chave precise ser apagada, essa técnica torna a remoção mais efetiva, pois dificulta a utilização de técnicas para sua recuperação. Mais detalhes podem ser vistos em: `http://clemens.endorphin.org/TKS1-draft.pdf`.

são essenciais no processo de montagem de um volume cifrado. O processo é complexo e segue uma série de etapas, as quais podem ser vistas em [57].

Vários algoritmos de cifragem de bloco simétricos podem ser utilizados, entre eles AES, Twofish, Serpent, Camellia e Kuznyechik. É possível fazer a cifragem dos blocos utilizando dois ou três algoritmos de forma encadeada. O modo de operação é o XTS[16]. Também usa a função de derivação PBKDF2 para gerar as chaves utilizadas na cifragem do cabeçalho e a chave secundária utilizada pelo modo de operação XTS.

No comparativo geral, os sistemas de armazenamento criptográficos baseados em blocos oferecem um nível maior de transparência em sua utilização. Quando utilizados para cifrar um disco inteiro também são úteis para manter a confidencialidade de partições de *swap* e de arquivos temporários. Por operarem em nível de blocos, abaixo dos sistemas de arquivos, conseguem manter a confidencialidade também dos metadados de arquivos e da estrutura de diretórios. Quando utilizados para cifrar um disco inteiro, escondem as informações sobre as partições contidas nele. São vantagens importantes quando se deseja dificultar a aplicação de técnicas forenses sobre discos. Operando no espaço do *kernel* e abaixo dos sistemas de arquivos, conseguem um desempenho ainda melhor se comparados aos sistemas de arquivos que rodam no espaço do *kernel*.

Suas desvantagens são: baixo controle no processo de cifragem, pois não permitem cifrar separadamente arquivos ou diretórios; geralmente necessitam de acesso como administrador do sistema para configuração e o acesso a múltiplos usuários é limitado.

A Tabela 3.1 sumariza os diferentes sistemas vistos neste capítulo. Ela também organiza esses sistemas considerando onde as funções de cifragem acontecem (espaço do usuário ou *kernel*) e também algumas características em comum refletindo a divisão das seções anteriormente descritas (aplicações; sistemas baseados no FUSE; basados no NFS; que rodam no espaço do *kernel*, interagindo diretamente com o VFS e dispositivos de blocos).

Na primeira coluna é apresentada uma classificação do desempenho dos sistemas. Essa classificação está diretamente relacionada ao espaço em que as funções de cifragem ocorrem e do quão próximas ao dispositivo final de armazenamento elas se encontram. A coluna "Nível de transparência" diz respeito à necessidade de interação do usuário final com o sistema para poder usufruir dos recursos de criptografia. Quanto maior o nível de transparência, menor a necessidade de interação. A unidade controlável de cifragem se refere à menor unidade cifrável a qual o usuário pode controlar. Também está diretamente ligada ao nível de transparência, sendo inversamente proporcional a ela, ou seja, quanto maior a transparência, menor o controle. As três colunas seguintes informam a capacidade do sistema de cifrar, além do conteúdo dos arquivos, outras informações importantes. A última coluna mostra a capacidade do sistema em oferecer, além do recurso de confidencialidade, o recurso da integridade.

A Tabela 3.2 sumariza os diferentes algoritmos de cifragem simétrica e os modos de operação suportados por todos os sistemas apresentados.

3.6 Demais sistemas criptográficos

Conforme descrito no início do capítulo, optou-se por descrever maiores detalhes somente dos sistemas criptográficos de código aberto. Porém, existem outras categorias que não foram descritas. Há várias soluções baseadas em software de código fechado. Por exemplo, no

[16]XTS é uma abreviação para *XEX-based Tweakable CodeBook mode with Ciphertext Stealing*. O XTS é um modo de operação criado especialmente para garantir a confidencialidade de dados gravados em dispositivos de armazenamento. Pode ser aplicado na cifragem de discos inteiros ou partições. Está definido no padrão IEEE 1619. Também passou a ser recomendado pelo NIST na SP 800-38E.

Tabela 3.1: Comparativo dos diversos sistemas criptográficos apresentados.

Espaço de funcionamento	Tipo	Sistema	Desempenho	Nível de transparência	Unidade controlável de cifragem	Cifra nomes de arquivos e diretórios	Cifra metadados e estrutura hierárquica	Cifra partições tmp e swap	Oferece também integridade
Espaço do usuário	Aplicações	Writer	Baixo	Baixo	Um único arquivo	Não	Não	Não	Não
		OpenSSL/ GPG	Baixo	Baixo	Um ou mais arquivos	Não	Não	Não	Sim
	FUSE	EncFS	Regular	Médio	Diretório montado	Sim	Não	Não	Sim
		CryFS	Regular	Médio	Diretório montado	Sim	Sim	Não	Sim
	NFS	CFS	Regular	Médio	Diretório montado	Sim	Não	Não	Não
Espaço do kernel		TCFS	Regular	Médio	Diretório montado e arquivos individuais	Sim	Não	Não	Sim
	Kernel/VFS	NCryptfs	Bom	Médio	Diretório montado	Sim	Não	Não	Não
		eCryptfs	Bom	Médio	Diretório montado e arquivos individuais	Sim	Não	Não	Sim
	Dispositivos de blocos	dm-crypt/ VeraCrypt	Ótimo	Alto	Partições e discos inteiros	Sim	Sim	Sim	Não

Tabela 3.2: Algoritmos de criptografia e modos de operação utilizados pelos sistemas criptográficos apresentados.

Sistema	Algoritmos de cifragem simétrica	Modos de operação
Writer	AES (padrão desde a versão 3.5)	CBC
OpenSSL	Vários	Vários
GPG	IDEA, 3DES, CAST5, Blowfish, AES, Twofish	Variação do CFB
EncFS	AES e Blowfish (OpenSSL)	CBC e CFB
CryFS	AES, MARS, Twofish, Serpent e CAST (Crypto++)	GCM e CFB
CFS	DES	ECB e OFB
TCFS	Permite escolher entre aqueles criados e disponibilizados como módulos no *kernel* e compatíveis com a interface do TCFS	CBC
NCryptfs	Inicialmente implementado usando Blowfish e AES. Permite escolher outros.	CFB
eCryptfs	Permite escolher os algoritmos disponíveis na CryptoAPI do Linux	CBC
dm-crypt	Permite escolher os algoritmos disponíveis na CryptoAPI do Linux	Permite escolher os modos disponíveis na Crypto API do Linux
VeraCrypt	AES, Twofish, Serpent, Camellia e Kuznyechik	XTS

sistema operacional Windows, há o *Encryption File System* (EFS) [58] que funciona integrado ao sistema de arquivos NTFS. Também da Microsoft, há uma solução para cifragem de discos chamada *BitLocker Drive Encryption* [59].

Existem também sistemas de arquivos distribuídos criptográficos. Além das características comuns de sistemas distribuídos (disponibilidade, desempenho, escalabilidade), também oferecem recursos de criptografia. São sistemas mais complexos que suportam múltiplos usuários e o compartilhamento de arquivos e diretórios cifrados entre eles. Além de cifragem simétrica, fazem uso da cifragem assimétrica e gerenciamento de chaves. Alguns exemplos desses sistemas são: *Self-Certifying File System* [60], Cepheus [61], *Secure Network-Attached Disks* (SNAD) [62], Plutus [63], SiRiUs [64]. Em [65] pode-se verificar uma pesquisa sobre esses sistemas, com uma análise resumida de seu funcionamento e recursos.

Há sistemas cujas funções de cifragem são realizadas por componentes de hardware integrados aos dispositivos de armazenamento. Essas funções são executadas no próprio hardware e não consomem recursos como memória e processamento do SO em que são utilizados. Nessa categoria se encaixam os dispositivos de armazenamento portáteis USB, gabinetes de disco (*disk enclosures*) e discos rígidos com suporte à criptografia. Uma abordagem sobre esses sistemas pode ser vista em [66].

3.7 Considerações finais

Como parte da fundamentação teórica, neste capítulo foram apresentados exemplos de sistemas de armazenamento com recursos de criptografia. Foi dado ênfase a sistemas de

código aberto. Esses sistemas foram apresentados na ordem das camadas de software na qual eles atuam.

No espaço do usuário foram apresentados exemplos de aplicações com recursos de criptografia (Seção 3.1) e dois sistemas de arquivos baseados no FUSE (EncFS e CryFS, Seção 3.2). Foi apresentado o CFS, baseado no NFS, mas cujas funções de cifragem também acontecem no espaço do usuário (Seção 3.3). No espaço do *kernel*, também baseado no NFS, foi apresentado o TCFS (também na Seção 3.3). Depois vieram os sistemas de arquivos que rodam no espaço do *kernel*. Foram apresentados o NCryptfs e o eCryptfs (Seção 3.4). Com relação a sistemas que interagem com a camada de E/S de blocos, foram apresentados o dm-crypt e o VeraCrypt, seguido de um resumo comparativo das características entre os sistemas apresentados (Seção 3.5).

Por último, como forma de complementação, foram brevemente apresentadas outras soluções de sistemas de armazenamento criptográficos que não se encaixam no perfil de sistemas de código aberto (Seção 3.6).

Os detalhes dos sistemas apresentados neste capítulo permitem saber em qual camada eles atuam e como aplicam suas funções de cifragem. São informações importantes que auxiliam no processo de alterá-los e integrá-los com bibliotecas e *frameworks* para processamento em GPU. Porém, também é essencial conhecer a arquitetura e a forma de programação das GPUs. Esse assunto será abordado no capítulo seguinte.

Capítulo 4

Arquitetura de GPUs NVIDIA e a plataforma CUDA

Por mais de duas décadas, o padrão na fabricação de processadores ficou concentrado em processadores com um único núcleo de processamento. A quantidade de transistores utilizados na sua construção continuou seguindo as previsões da lei de Moore, onde a utilização de transistores nos chips praticamente dobra a cada dois anos. Esse fato se refletia na criação de processadores cada vez mais complexos, com capacidades e velocidades maiores de processamento [67].

Porém, desde 2003, devido a questões de consumo de energia e dissipação de calor, tornou-se mais difícil criar processadores com frequências cada vez maiores. Praticamente todas os fabricantes mudaram para um modelo onde, para se conseguir um poder de processamento maior, são integrados dois ou mais núcleos de processamento num único chip [3].

As indústrias de semicondutores seguiram dois caminhos distintos: a criação de processadores com múltiplos núcleos (*multicores*) e a criação de processadores para lidar com grandes quantidades de *threads* (*many-threads*) [68].

No primeiro caso, iniciou-se com chips que continham dois núcleos de processamento e atualmente podem ter até 24 núcleos e suporte para execução simultânea de 48 *threads*[1]. Tal arquitetura dedica boa parte da área do chip para estruturas sofisticadas de controle lógico. Outra parte considerável da área do chip é ocupada com memórias *cache* de grande capacidade. A utilização desses componentes visam reduzir a latência de execução de uma única *thread*. Por isso, processadores com essa arquitetura são denominados orientados à latência [68].

No segundo caso, houve o desenvolvimento de arquiteturas que tinham por objetivo maximizar a vazão de dados, visando suprir as necessidades de aplicações específicas. São aplicações com um alto nível de paralelismo no processamento de dados, usando grandes quantidades de *threads*. Um exemplo clássico são as placas gráficas e suas unidades de processamento gráfico (GPU[2]). Modelos mais antigos possuíam a capacidade de executar algumas centenas de *threads*. Modelos mais novos podem executar até 3.584 *threads* simultaneamente[3].

Na arquitetura das GPUs, ao contrário da arquitetura tradicional das CPUs[4], a maior parte da área disponível em chip é utilizada para abrigar os núcleos de processamento. A memória principal da GPU é integrada ao dispositivo e oferece altas taxas de largura de banda. São

[1]Capacidade do processador Intel Xeon E7 (http://www.intel.com.br/content/www/br/pt/products/ processors/xeon/e7-processors/e7-8890-v4.html).

[2]Sigla do inglês *Graphics Processing Unit*.

[3]Capacidade do modelo NVIDIA Tesla P100 (http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-datasheet.pdf).

[4]Siga do inglês *Central Processing Unit*.

características que permitem a execução de uma grande quantidade de *threads* simultaneamente, dando a essa arquitetura a denominação de orientada à vazão de dados [68].

Inicialmente as GPUs eram utilizadas com a finalidade de acelerar o processamento gráfico para jogos eletrônicos. As duas linguagens existentes, OpenGL e Direct3D, ofereciam uma API[5] específica para processamento gráfico. Alguns desenvolvedores faziam uso dessa API para ter acesso aos núcleos de processamento da GPU e executar outros tipos de aplicação. Esse uso passou a ser denominado programação de propósito geral utilizado unidades de processamento gráfico (GPGPU[6]) [68]. Porém, a programação de aplicações não gráficas utilizando essa API era complexa.

Esse cenário de complexidade começou a mudar em 2007, quando a NVIDIA lançou a GPU G80 e a plataforma CUDA. A linguagem CUDA C foi disponibilizada e uma API que facilitava o acesso aos recursos de processamento na GPU. A partir desse ponto, tornou-se comum o uso da plataforma CUDA e do processamento em GPU para se alcançar níveis significativos de aceleração em diversas aplicações.

Na Seção 4.1 são vistos detalhes gerais das arquiteturas de GPUs NVIDIA. A Seção 4.2 apresenta a plataforma CUDA e seu funcionamento nessas GPUs. A Seção 4.3 oferece uma breve visão e características dos diferentes tipos de memórias disponíveis. A Seção 4.4 apresenta o recurso de *streams* oferecido pela plataforma CUDA, a qual permite executar tarefas de forma paralela. Por fim, a Seção 4.5 apresenta uma visão geral de GPUs pertencentes a outras fabricantes, bem como plataformas alternativas de programação para GPUs.

4.1 Arquitetura de GPUs NVIDIA

Umas das classificações da arquitetura de computadores é de acordo com o modo como as instruções são aplicadas sobre os dados. Segundo a taxonomia de Flynn, essa classificação pode ser feita em quatro modos: (i) Única instrução e único dado (SISD[7]); (ii) Única instrução e múltiplos dados (SIMD); (iii) Múltiplas instruções e único dado (MISD) e (iv) Múltiplas instruções e múltiplos dados (MIMD) [69].

No modo SISD, há apenas um único fluxo de instruções que são aplicadas a um único fluxo de dados. No modo SIMD, um único fluxo de instruções é aplicado a múltiplos fluxos de dados. Nesse caso múltiplos processadores são necessários para executar essas instruções simultaneamente. No modo MISD, há múltiplos processadores que aplicam fluxos de instruções distintas num único fluxo de dados. No modo MIMD, há múltiplos processadores que conseguem executar operações distintas de um fluxo de instruções em múltiplos fluxos de dados.

Também é possível classificar a arquitetura de computadores de acordo com a organização de sua memória. Há dois modelos: memória compartilhada e memória distribuída [69]. No modelo de memória distribuída, há vários computadores, cada um com sua própria memória local e espaço de endereçamento distinto. O compartilhamento do conteúdo de suas memórias ocorre através de mensagens. Esse modelo é conhecido como multicomputadores. No modelo de memória compartilhada, os processadores compartilham o mesmo espaço de endereçamento de memória e são conhecidos como multiprocessadores.

[5]Sigla do inglês *Application Program Interface*.

[6]Sigla do inglês *General-purpose Programming using a Graphics Processing Unit*.

[7]As siglas correspondem a expressões em inglês. São elas: *Single Instruction Single Data* (SISD); *Single Instruction Multiple Data*(SIMD); *Multiple Instruction Single Data* (MISD) e *Multiple Instruction Multiple Data* (MIMD).

Em sistemas multiprocessados, quando os processadores estão reunidos num único chip, recebem e denominação de *multicore*. O termo *many-core* é utilizado para nomear arquiteturas *multicore* que possuem um grande número de núcleos de processamento [69].

As GPUs NVIDIA são exemplos de arquiteturas *many-core* que operam em um modo similar ao SIMD. Possuem alta capacidade de *multithreading*, bem como paralelismo em nível de instrução. Essa alta capacidade de *multithreading* está diretamente relacionada a grande quantidade de processadores, chamados CUDA *cores*, reunidos numa única GPU.

Ao longo de sua evolução, as GPUs NVIDIA com suporte a plataforma CUDA passaram por diferentes arquiteturas, recebendo nomes distintos: G80 (2006), GT200 (2008), Fermi (2010), Kepler (2012), Maxwell (2014), Pascal (2016) e Volta (2017). Cada GPU possui uma capacidade de computação[8] (CC) diferente, representada por um número no formato $X.Y$. A letra X corresponde a um número associado à arquitetura principal e a letra Y a um número que indica um melhoramento sobre ela.

O componente principal das GPUs NVIDIA são os chamados *Streamming Multiprocessors* (SM). Os subcomponentes de cada SM podem ser vistos na Figura 4.1. Na sequência, é apresentada uma breve descrição de suas funções e algumas de suas características físicas, como quantidades e capacidades. Essas características podem variar de acordo com a arquitetura e CC. As características descritas na sequência são específicas da arquitetura Pascal, CC 6.0 [70].

Figura 4.1: Componentes de um SM da arquitetura Pascal 6.0 [70].

Cada SM possui 64 núcleos (*cores*). Internamente, cada núcleo possui uma unidade de aritmética e lógica e uma unidade de ponto flutuante com precisão simples (32 bits). Há 32 unidades de ponto flutuante (DP *Unit*[9]) de precisão dupla (64 bits). O controle de endereçamento para carregamento e armazenamento de dados é feito por 16 unidades (LD/ST[10]) capazes de realizar ambas as operações simultaneamente. Há 16 unidades para funções especiais (SFU[11])

[8]Tradução do termo em inglês "*compute capability*".
[9]Sigla do inglês *Double Precision Unit*.
[10]Sigla do inglês *Load/Store*.
[11]Sigla do inglês *Special Function Unit*.

capazes de realizar operações como seno, cosseno e raiz quadrada. Os núcleos, DP *units*, LD/STs e SFUs são denominadas unidades funcionais.

Uma unidade especial da GPU é responsável por dividir as *threads* em blocos e atribuí-las aos SMs da GPU. As *threads* dentro de um bloco são executadas concorrentemente em um SM. Quando blocos de *threads* terminam sua execução no SM, novos blocos são atribuídos a ele. Essa forma automática de distribuição dos blocos entre os SMs e sua execução independente cria um ambiente de escalabilidade transparente. Na Figura 4.2 está ilustrado um exemplo onde uma aplicação CUDA é executada em duas GPUs diferentes. Sem precisar fazer alterações na aplicação, ela é executada em tempo menor na GPU com mais recursos, ou seja, a GPU com quatro SMs.

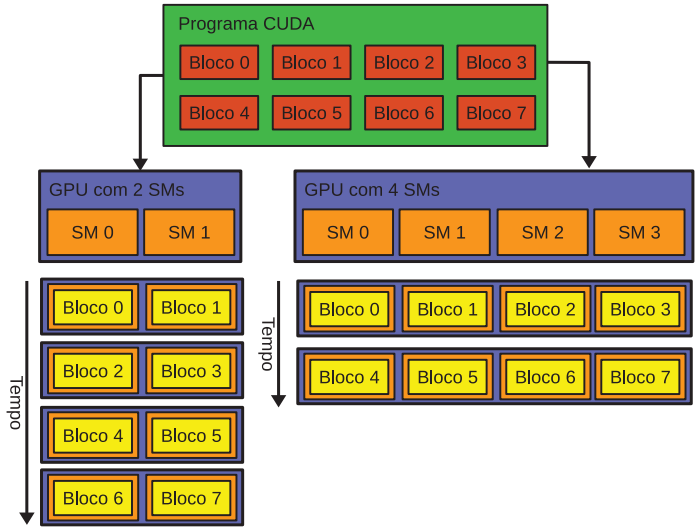


Figura 4.2: Escalabilidade transparente em GPUs de capacidades diferentes. Adaptado de [71].

Dentro do SM, as *threads* são reunidas em grupos de 32 *threads*, denominados *warps*. Dois escalonadores de *warps*[12] e quatro unidades de despacho[13] são responsáveis pelo escalonamento dos *warps* e execução das instruções em cada *thread*, fazendo uso das unidades funcionais. A dupla unidade de despacho por escalonador de *warp* permite que sejam executadas simultaneamente duas instruções por *warp*. O duplo escalonador permite que sejam executados simultaneamente dois *warps* por SM.

Cada SM possui um conjunto interno de memórias formado por registradores e pela *shared memory*. Essas memórias oferecem baixas latências e altas larguras de banda. Para leitura e gravação, com acesso privado a cada *thread*, há 65.536 registradores[14]. Também para leitura e gravação, porém utilizada para comunicação entre *threads* de um mesmo bloco, há 64 KB de *shared memory*. Há uma hierarquia de *caches* que auxiliam a reduzir a latência e aumentar a vazão de dados ao acessar dados das memórias externas ao chip[15]. Essas memórias externas são a *global memory*, *local memory*, *constant memory* e *texture memory*. Maiores detalhes sobre os diferentes tipos de memória estão descritos na Seção 4.3.

Quando um bloco de *threads* é atribuído a um SM, para se tornar ativo e ser efetivamente executado, é necessário que sejam alocados recursos como registradores e *shared memory*. Esses

[12]Identificados como "*warp scheduler*", na Figura 4.1.
[13]Identificadas como "*dispatch unit*", na Figura 4.1.
[14]Identificados como "*Register File*", na Figura 4.1.
[15]Há uma memória *cache* de nível 1 (24 KB) interna ao SM, identificada como "Texture/L1 cache", na Figura 4.1. Também há uma memória *cache* de nível 2 (4 MB), não ilustrada na Figura 4.1. Essa memória está inserida dentro do chip, porém fora do SM, compartilhada por todos os SMs da GPU.

recursos são limitados e geralmente a quantidade necessária de registradores e uso da *shared memory* ditam a quantidade máxima de blocos residentes em cada SM. Também, de acordo com a CC da GPU, há limites máximos na quantidade de blocos, *threads* e *warps* por SM.

Quando um bloco é executado, os *warps* contidos nele podem assumir três estados: selecionado, parado ou elegível. Um *warp* se torna elegível quando todos os operandos necessários para a execução de uma instrução estão carregados. Se houver 32 núcleos ociosos no SM, esse *warp* pode ser selecionado para execução, passando para o estado selecionado. Caso o *warp* selecionado venha a executar uma instrução de latência maior, por exemplo, o carregamento de outro dado da memória global, sua execução é interrompida. Seu estado é alterado para parado e permanece assim até que o dado necessário esteja disponível, quando novamente voltará ao estado de elegível.

Quando um bloco se torna ativo, os recursos de hardware necessários para o controle do contexto de execução das *threads* já são divididos entre elas. Isso permite que a troca do contexto de execução entre *warps* aconteça praticamente sem sobrecarga. Ao passar para o estado parado, os núcleos de processamento utilizados pelo *warp* são liberados, podendo ser utilizados por outro *warp* selecionado. Essa troca sem sobrecarga de execução de *warps* e a forma como elas são escalonadas permitem esconder a latência de instruções mais demoradas. Mantendo-se um determinado número de *warps* no estado elegível, os recursos de processamento são utilizados ao máximo, permitindo-se alcançar altos níveis de execução paralela de *threads*.

A forma como as instruções são atribuídas às *threads* de um *warp* selecionado é similar ao modo SIMD. Todas as *threads* pertencentes ao mesmo *warp* executam a mesma instrução. Porém, há algumas diferenças. Cada *thread* possui o seu próprio contador de endereço de instrução e registrador de estado. Assim, cada *thread* pode seguir um caminho diferente na execução. Essas diferenças caracterizam um modelo denominado pela NVIDIA de única instrução e múltiplas *threads* (SIMT[16]).

A forma como as instruções são atribuídas aos *warps* e a forma como eles são escalonados ao longo do tempo estão ilustradas na Figura 4.3. Nela é possível visualizar um dos escalonadores de *warps* e sua a dupla unidade de despacho que permite atribuir duas instruções diferentes ao mesmo *warp*.

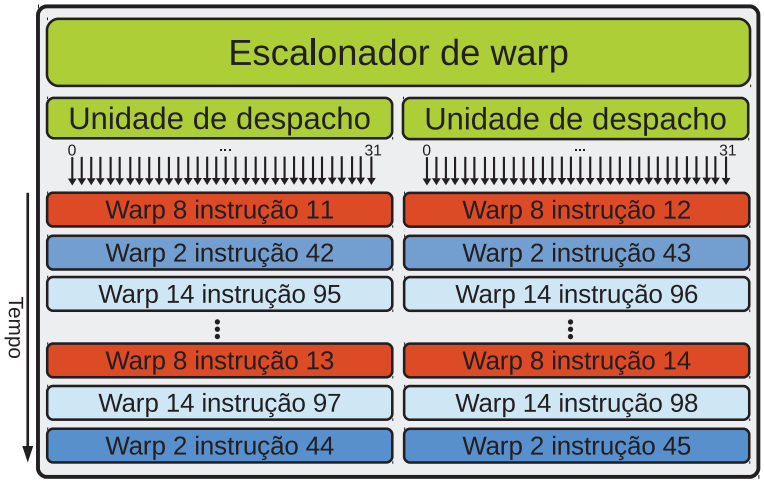


Figura 4.3: Funcionamento do escalonador de *warps*. Adaptado de [72].

[16]Sigla do inglês *Single Instruction Multiple Thread*.

Quando *threads* de um mesmo *warp* seguem caminhos diferentes de execução, acontece o que é denominado divergência de *warp*[17]. Isso ocasiona uma serialização na execução: primeiro são executadas as *threads* de um caminho, depois as *threads* do caminho diferente. Isso leva a uma subutilização dos recursos de processamento, pois enquanto as *threads* que não estão sendo executadas ficam desativadas, as unidades funcionais utilizadas por elas ficam ociosas.

Um exemplo de divergência pode ser visto na Figura 4.4. Quando as *threads* encontram a cláusula `if`, somente aquelas cujos índices sejam menor do que quatro executam as instruções *A* e *B*. As *threads* de quatro a 31 ficam desativadas. Quando as *threads* de zero a três terminam de executar suas instruções, elas são desativadas e as *threads* de quatro a 31 executam as instruções *X* e *Y*. Somente após essas etapas elas podem convergir e seguir executando a instrução *Z*.

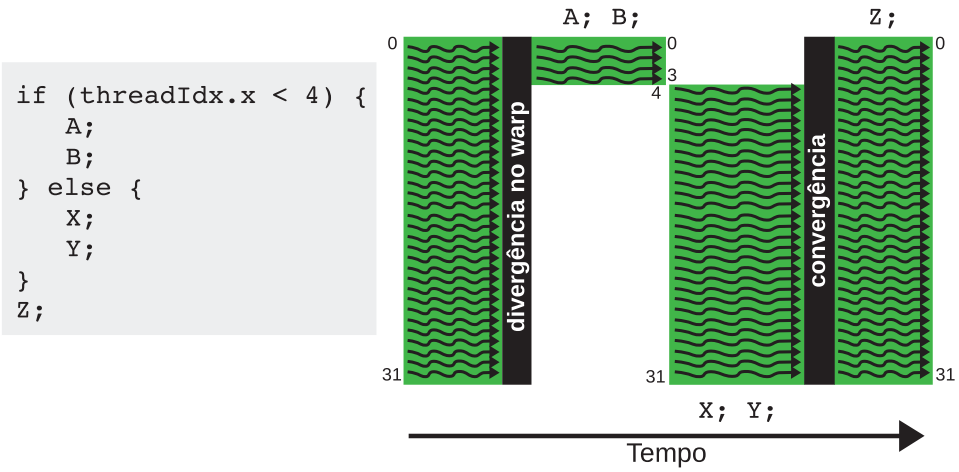


Figura 4.4: Exemplo de divergência na execução de um *warp*. Adaptado de [73].

Com relação à memória situada na parte externa do chip dos SMs, denominada *global memory*, até a arquitetura Maxwell, elas são do tipo GDDR5[18]. Dependendo do modelo, oferecem uma largura de banda máxima teórica por volta de 540 GB/s[19], com capacidade total de 12 GB. A partir da arquitetura Pascal, alguns modelos passaram a utilizar uma nova tecnologia de memórias empilháveis denominada HBM2[20]. A largura de banda máxima oferecida é de 732 GB/s, com capacidade total de 16 GB[21].

Até a arquitetura Maxwell, a forma de conexão entre a memória da GPU e da CPU era através do barramento PCI Express (PCIe). Na versão PCIe 3.0, o limite teórico máximo de banda é 16 GB/s. Essa banda, se comparada com a banda disponível entre a memória de sistema e a CPU (50 GB/s[22]) e da GPU e seus SMs (732 GB/s), é um gargalo conhecido na computação heterogênea utilizando CPUs e GPUs [68][69].

Na arquitetura Pascal, uma nova tecnologia denominada NVLink foi criada. É possível utilizar até quatro conexões NVLink por GPU. Cada conexão oferece uma largura de banda teórica máxima de 40 GB/s. Sua principal utilidade é realizar interconexões entre GPUs. Porém,

[17]Tradução do termo em inglês "*warp divergence*".

[18]Sigla do inglês *Double Data Rate type five synchronous Graphics Random-access memory*.

[19]Para o modelo Titan Xp (https://www.nvidia.com/en-us/geforce/products/10series/titan-xp/).

[20]Sigla do inglês *High Bandwidth Memory*, segunda geração.

[21]Modelo Tesla P100 (http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-datasheet.pdf).

[22]Em memórias do tipo DDR4 atuando em *dual channel*.

para processadores que suportem tal tecnologia, é possível agregar as conexões, atingindo uma banda de até 160 GB/s entre a CPU e a GPU [70].

Para mais detalhes sobre a arquitetura de cada geração das GPUs, pode-se consultar os *white papers*: Fermi [74], Kepler [72], Maxwell [75], Pascal [70] e Volta [73]. Mais detalhes sobre as diferentes características das CCs, podem ser vistos em [69].

4.2 Plataforma CUDA

Segundo [69], CUDA consiste em uma plataforma e um modelo de programação de uso geral que permite utilizar os recursos de hardware de GPUs NVIDIA com a finalidade de executar processamento paralelo de dados. A GPU é controlada por um driver de dispositivo e a comunicação com esse driver é feita através do *runtime* CUDA. Em uma aplicação, o controle desse *runtime* é feito através do uso de um conjunto específico de bibliotecas e APIs. Para o desenvolvimento de aplicações, são utilizadas extensões para linguagens de programação conhecidas (C/C++, Fortran), *wrappers* para Java e Python, linguagens baseadas em diretivas de compilação como OpenACC, etc.

A extensão para a linguagem de programação C/C++ é denominada CUDA C. Além de oferecer uma biblioteca com APIs que permitem acessar os recursos das GPUs, essa extensão adiciona um conjunto de palavras-chave utilizadas para identificar trechos de código que deverão ser executados na GPU. Assim, um programa CUDA C permite criar aplicações com recursos de computação heterogênea. Parte do código é escrito para ser executado na CPU[23] e parte do código para ser executado na GPU[24].

Um compilador específico, denominado nvcc, é utilizado para compilar um programa escrito em CUDA C. Ele separa o trecho de código que deve ser executado na GPU e gera um código intermediário denominado PTX ou um código binário final denominado cubin. Os trechos de código que devem ser executados na CPU são repassados a compiladores C/C++ tradicionais. Quando a aplicação é executada pela primeira vez e for utilizado o código PTX, o *runtime* CUDA realiza mais um passo de compilação do tipo *just-in-time* para a geração do código binário final que será executado na GPU.

Em CUDA C, as funções principais que devem ser executadas na GPU são denominadas *kernels*. Quando essas funções são chamadas, elas são executadas na GPU N vezes, por N diferentes *threads* em paralelo.

Umas das abstrações oferecidas pela plataforma CUDA é uma hierarquia de *threads*. Quando um *kernel* é lançado, é possível definir a quantidade e a forma de agrupamento das *threads* que executarão o código definido no *kernel*. As *threads* podem ser agrupadas em blocos, chamados de blocos de *threads*, ou simplesmente blocos. Os blocos podem ser agrupados em grades. Ambos os blocos e grades podem ser definidos em uma, duas ou três dimensões.

A Figura 4.5 ilustra o lançamento de um *kernel* onde são utilizadas apenas duas dimensões tanto para a grade quanto para os blocos. A grade possui seis blocos e cada bloco possui 15 *threads*. Considerando a quantidade total de blocos e *threads* por bloco, esse *kernel* será lançado utilizando 90 *threads*.

É responsabilidade do programador definir a dimensão e o tamanho da grade e dos blocos. Nesse aspecto, há uma certa flexibilidade e geralmente se leva em consideração características dos dados a serem processados. Há várias questões envolvidas nessa definição, além da simples quantidade total de *threads* utilizadas. Segundo [68], [69] e [71], pode-se destacar:

[23] Também é utilizado o termo "*host*" para se referir à CPU e à memória de sistema.

[24] Também é utilizado o termo "dispositivo" para se referir à GPU e seu conjunto de memórias.

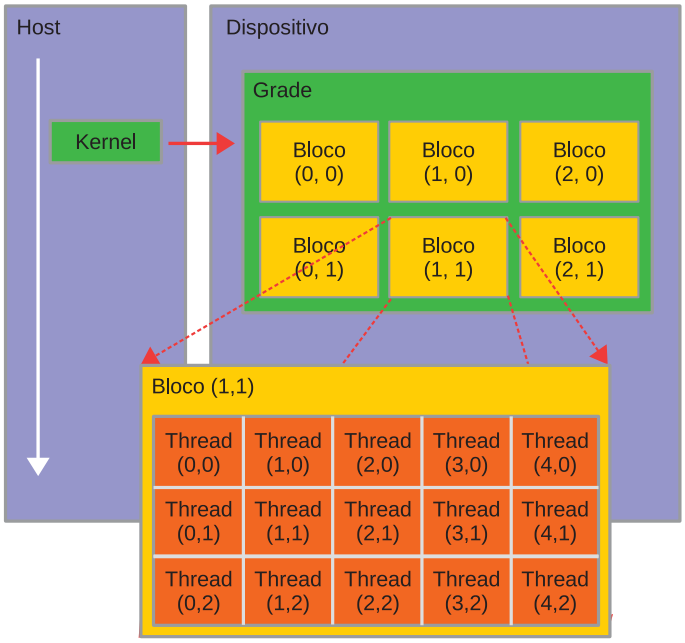


Figura 4.5: Organização das *threads* em grade e blocos. Adaptado de [69].

- *Dimensionalidade dos dados*: de acordo com os dados a serem processados, definir a dimensão das grades e dos blocos. Por exemplo, o processamento de um vetor normalmente é mapeado para uma grade e blocos de uma única dimensão. O processamento de imagens bidimensionais é naturalmente mapeado para grades e blocos bidimensionais;
- *Limites de hardware*: de acordo com a CC da GPU, há valores máximos para o tamanho dos blocos e das grades. Esses valores podem ser consultados dinamicamente através de funções específicas da API CUDA;
- *Utilização de recursos*: recursos como registradores e *shared memory* precisam ser alocados quando cada bloco é atribuído a um SM. Esses recursos são limitados e quanto maior sua utilização, menos blocos podem ser alocados por SM, reduzindo sua taxa de ocupação;
- *Tamanho múltiplo do warp*: é relevante utilizar tamanhos de bloco que sejam múltiplos do tamanho do *warp*. Assim, diminui-se a quantidade de *threads* que ficam desativadas por estarem em uma região não válida para processar dados.

Essas e outras questões têm um impacto direto na taxa de ocupação dos SMs e consequentemente no desempenho geral do *kernel* ao ser executado na GPU. Maiores detalhes podem ser vistos em [68], [69] e [71].

A implementação de uma aplicação CUDA geralmente segue os seguintes passos: (i) alocação de memória na GPU; (ii) cópia dos dados a serem processados da memória da CPU para a memória da GPU; (iii) chamada do *kernel* que executará o processamento na GPU; (iv) cópia dos dados processados da memória da GPU para a memória da CPU e (v) liberação da memória alocada na GPU. Esses passos podem ser vistos na aplicação exemplo descrita no Quadro 4.1.

A aplicação faz a adição de dois vetores de tamanhos iguais (16.777.216 elementos cada um) de forma paralela na GPU. Os passos da alocação de memória na GPU podem ser vistos nas linhas 18 à 20, através da utilização da função `cudaMalloc()`. Nas linhas 23 e 24 a

```
1  #include <cuda_runtime.h>
2  #include <stdio.h>
3
4  // Definição do kernel
5  __global__ void somaVetorGPU(float *A, float *B, float *C, unsigned int
       qtdeElementos) {
6        unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
7        C[i] = A[i] + B[i];
8  }
9
10 int main(int argc, char **argv) {
11       unsigned int qtdeElementos = 16777216;
12       unsigned int qtdeBytes = qtdeElementos * sizeof(float);
13       // ... Omitidas rotinas de criação, inicialização e alocação de
14       // memória para variáveis no host, bem como a inicialização dos
15       // vetores h_A e h_B ...
16
17       // Aloca memória para vetores na GPU
18       cudaMalloc((float**)&d_A, qtdeBytes);
19       cudaMalloc((float**)&d_B, qtdeBytes);
20       cudaMalloc((float**)&d_C, qtdeBytes);
21
22       // Copia dados da CPU pra GPU
23       cudaMemcpy(d_A, h_A, qtdeBytes, cudaMemcpyHostToDevice);
24       cudaMemcpy(d_B, h_B, qtdeBytes, cudaMemcpyHostToDevice);
25
26       // Definição do tamanho do bloco e da grade
27       int qtdeThreadsPorBloco = 512;
28       dim3 tamanhoBloco(qtdeThreadsPorBloco);
29       dim3 tamanhoGrade(qtdeElementos / qtdeThreadsPorBloco);
30
31       // Chama o kernel a ser executado na GPU
32       somaVetorGPU<<<tamanhoGrade, tamanhoBloco>>> (d_A, d_B, d_C);
33       cudaDeviceSynchronize();
34
35       // Copia dados da GPU pra CPU
36       cudaMemcpy(h_C, d_C, qtdeBytes, cudaMemcpyDeviceToHost);
37
38       // Libera memória da GPU
39       cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
40
41       // ... Usa valores dos vetores somados que agora estão
42       // acessíveis no vetor h_C ...
43       // ... aplicação continua ...
44 }
```

Quadro 4.1: Exemplo de aplicação para soma de vetores escrita em CUDA C.

função `cudaMemcpy()` é utilizada para copiar os dados do vetor `h_A` e `h_B` da memória da CPU para a memória da GPU.

Nas linhas 27 à 29 são definidos os tamanhos do bloco e da grade. Apenas uma dimensão é utilizada. A quantidade de blocos necessários é determinada pela divisão da quantidade de elementos (`qtdeElementos`) pelo número de *threads* utilizadas em cada bloco

(`qtdeThreadsPorBloco`). Como o tamanho do bloco foi definido como sendo 512, serão necessários 32.768 blocos de *threads*.

A chamada do *kernel* acontece na linha 32. Entre os símbolos "`<<< >>>`" são informados os tamanhos da grade e dos blocos. Como parâmetro são passados os ponteiros dos vetores na memória da GPU.

O *kernel* que será executado na GPU e fará a soma dos vetores paralelamente pode ser visto nas linhas 5 à 8. Pode-se verificar a utilização do qualificador "`__global__`" para identificar a função que será utilizada como *kernel*. Cada *thread* executará as mesmas instruções contidas nele. Na linha 6 é calculada a posição dos dados nos vetores sobre os quais cada *thread* vai trabalhar. As variáveis `blockIdx`, `blockDim` e `threadIdx` são variáveis criadas automaticamente pelo *runtime* CUDA e possuem o índice do bloco na grade, o tamanho do bloco e o índice da *thread* no bloco, respectivamente. A soma realizada por cada *thread*, sobre seus respectivos pares de elementos dos vetores, é feita na linha 7 e o resultado é armazenado no vetor `C`.

Na linha 33 é executada a função `cudaDeviceSynchronize()` visando bloquear a execução de código no *host* até que a execução do *kernel* na GPU termine. Após a sincronização, são copiados os dados processados da GPU para a CPU (linha 36). A função utilizada para copiar é a mesma já utilizada anteriormente. Altera-se somente o sentido da cópia, modificando-se o último parâmetro da função.

Após feita a cópia dos dados da GPU para a CPU, eles podem ser utilizados normalmente na aplicação. A última etapa, correspondente à liberação da memória alocada na GPU, pode ser vista na linha 39.

Uma aplicação CUDA não precisa seguir todas as etapas anteriormente descritas. Existem recursos e funções mais avançados da plataforma CUDA para realizar o gerenciamento de memória e controle da cópia de dados. Incluindo cópias entre CPU e GPU, entre duas ou mais GPUs e até mesmo entre outros dispositivos e a GPU. Alguns desses recursos são conhecidos como "*zero-copy memory*", "*unified virtual addressing*", "*unified memory*" e GPUDirect. Por exemplo, utilizando-se o recurso de "*unified memory*", torna-se desnecessário controlar explicitamente a cópia de dados entre a CPU e a GPU. Maiores detalhes podem ser vistos em [68], [69] e [71].

4.3 Hierarquia de memórias

A plataforma CUDA também oferece um controle refinado na utilização dos diferentes tipos de memória existentes na GPU. Essas memórias estão organizadas numa hierarquia, possuindo características distintas com relação a latência de acesso e largura de banda. Há memórias não programáveis, dedicadas às funções de *cache*, como a memória L1, L2, *texture* e *constant cache*. Há memórias que podem ser controladas diretamente pelo programador, como os registradores, *shared memory*, *local memory*, *constant memory*, *texture memory* e *global memory* [69].

A organização das memórias programáveis pode ser vista na Figura 4.6. As setas entre os diferentes tipos de memória indicam o escopo de acesso. As memórias contidas no quadro laranja ficam dentro do SM (*on-chip*) e as demais estão contidas na GPU, porém fora do SM (*off-chip*).

Os registradores são a memória de acesso mais rápido. Variáveis declaradas sem nenhum tipo de qualificador adicional, dentro do *kernel*, geralmente são alocadas nos registradores. Vetores cujos índices são constantes e que possam ser determinados em tempo de compilação também são armazenados neles. A exceção são *structures*, vetores grandes ou qualquer variável

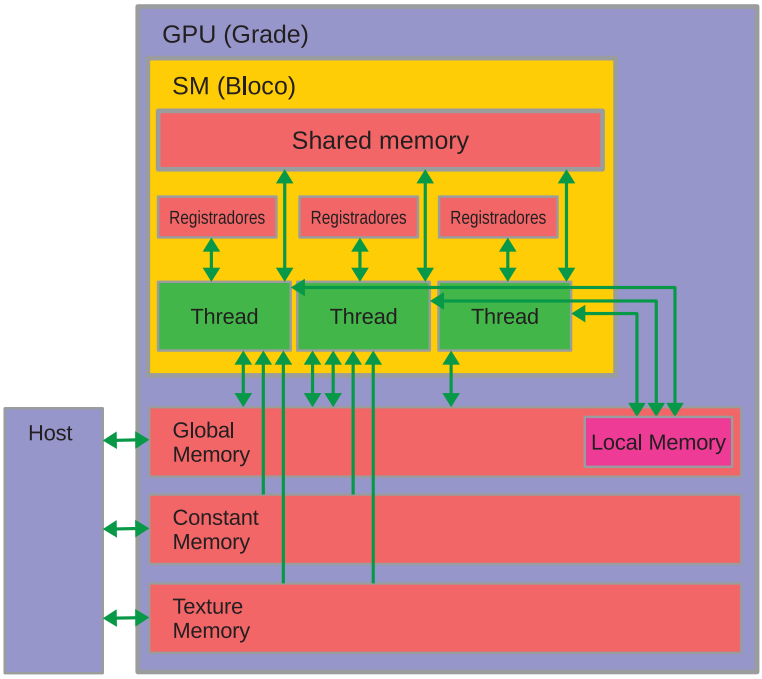


Figura 4.6: Organização e escopo de acesso dos diversos tipos de memória da GPU. Adaptado de [69].

caso a utilização de registradores pelo *kernel* seja muito alta. O escopo de acesso aos registradores é privado a cada *thread* e, a partir da arquitetura Kepler, pode ser compartilhado entre *threads* de um mesmo *warp* através do uso de uma instrução especial. Dados armazenados em registradores estão disponíveis enquanto o bloco estiver ativo. Esse tempo em que os dados estão disponíveis para acesso é chamado de tempo de vida.

A quantidade de registradores por SM é limitada[25], assim como é limitada a quantidade máxima de registradores por *thread*[26]. Apesar de serem valores significativos, é importante observar que os registradores são um dos recursos que precisam ser alocados antecipadamente pelos blocos ativos. Portanto, ao se manter a utilização dos registradores baixa, é possível se atribuir mais blocos por SM e consequentemente se obter níveis maiores de paralelismo.

A *local memory* fica fora do chip do SM e possui os mesmos tempos de acesso e restrições de desempenho da *global memory*. Nela são armazenadas as seguintes variáveis: vetores cujos índices não podem ser determinados em tempo de compilação; *structures* e vetores grandes; demais variáveis que não podem ser alocadas em registradores devido a elevada quantidade de registradores já utilizados pelo *kernel*. Os dados contidos nela podem sofrer ação de *cache* nas memórias L1 e L2. O escopo de acesso e tempo de vida são os mesmos dos registradores.

Depois dos registradores, a *shared memory* é a que oferece os melhores tempos de acesso. Assim como os registradores, está contida dentro do chip do SM. É uma das formas mais eficientes de troca de dados entre *threads* de um mesmo bloco. Seu escopo de acesso é dentro do bloco e o tempo de vida corresponde ao tempo em que o bloco permanecer ativo. Também é um recurso relativamente limitado[27].

[25] Na maioria das arquiteturas o valor máximo é de 65.536.
[26] Na arquitetura Pascal, esse valor é de 255.
[27] Na GPU P100, arquitetura Pascal, há 64 KB de *shared memory*.

Dados armazenados na *shared memory* podem ser acessados de uma forma mais flexível. Não é necessário seguir os padrões de acesso de memória agrupada[28] para se obter um bom desempenho, como acontece com a *global memory*. Fisicamente a *shared memory* está disposta em 32 bancos de memória que podem ser acessados simultaneamente. É necessário haver um cuidado com relação a como o acesso é feito pelas *threads* dentro de um mesmo *warp*. Esse cuidado visa evitar um problema chamado de "conflito de banco"[29], o qual ocasiona queda no desempenho.

A *constant memory* fica fora do chip e é destinada a armazenar valores que se mantenham estáticos durante a execução do *kernel*. Dados acessados nela sofrem ação de *cache* em uma memória *cache* interna específica, chamada *constant cache*. Seu uso é indicado principalmente quando as *threads* de um mesmo *warp* precisam acessar um valor estático de um mesmo endereço. Dessa forma, o valor acessado é entregue às *threads* através de uma operação de *broadcast*. O escopo de acesso inclui todas as *threads* de todos os *kernels* lançados por uma aplicação, bem como a aplicação em si que roda na CPU. Os dados armazenados nela persistem enquanto durar a aplicação.

Quando o acesso precisa ser feito a diferentes endereços de memória, os quais possuam uma certa proximidade espacial, o indicado é a utilização da *texture memory*. Também é uma memória de acesso somente para leitura e a qual possui memória *cache* interna exclusiva, a *texture cache*. Permite realizar algumas operações especiais como: conversão automática entre alguns tipos de dados; utilizar formas diferentes de indexação para acesso a vetores; realizar operações de interpolação de valores que estejam em posições próximas[30]. O escopo de acesso e o tempo de vida são os mesmos da *constant memory*.

A *global memory* fica fora do chip do SM. É a memória de maior capacidade na GPU. Com relação aos demais tipos de memória, possui maior latência e menor largura de banda. Parte disso é compensado pelo fato dos dados acessados nela sofrerem ação das memórias *cache* L1 e L2. A *global memory* é uma das principais formas de compartilhamento de dados entre diferentes *kernels* de uma mesma aplicação. O escopo de acesso são os mesmos da *constant* e *texture memory*. O tempo de vida é controlado pela aplicação através do uso de funções específicas da API CUDA para alocação e liberação de memória.

O acesso à *global memory* é feito com transações de memória de tamanhos variados, sendo os dois mais importantes o de 32 e 128 bytes. Transações de 32 bytes alinham-se com a *cache* L2 e transações de 128 bytes com a *cache* L1. Para haver uma relação alta entre dados carregados e dados processados é importante que o primeiro endereço acessado seja um múltiplo da granularidade dos *caches*, ou seja, 32 bytes para a L2 e 128 bytes para a L1.

Outro aspecto importante é com relação às regiões de memória acessadas. É importante que *threads* de um mesmo *warp* acessem regiões de memória que sejam fisicamente contíguas. Isso permite que várias transações de memória possam ser combinadas numa quantidade menor de transações, sendo carregados mais bytes por transação. Deve-se evitar o acesso espalhado por várias regiões distantes ou o acesso de forma intercalada[31].

Um resumo dos diferentes tipos de memória programáveis com algumas de suas características pode ser visto na Tabela 4.1.

[28] Tradução do termo em inglês "*coalesced*".

[29] Tradução do termo em inglês "*bank conflict*".

[30] Essas operações são realizadas por unidades funcionais específicas dentro do SM. Na Figura 4.1, essas unidades estão identificadas como "Tex".

[31] Tradução do termo em inglês "*strided access*".

Tabela 4.1: Algumas características dos diferentes tipos de memória da GPU.

Tipo de memória	Localização no chip	Tipo de acesso	Escopo de acesso	Tempo de vida
Registradores	Dentro	Leitura e gravação	Por *thread*	Enquanto bloco estiver ativo
Shared	Dentro	Leitura e gravação	*Threads* no bloco	Enquanto bloco estiver ativo
Local	Fora	Leitura e gravação	Por *thread*	Enquanto bloco estiver ativo
Constant e Texture	Fora	Somente leitura	Todas as *threads* + aplicação	Duração da aplicação
Global	Fora	Leitura e gravação	Todas as *threads* + aplicação	Controlado pela aplicação

4.4 Utilização de *streams*

Além do paralelismo em nível de *threads*, a plataforma CUDA também permite realizar o paralelismo em nível de tarefas. Esse tipo de paralelismo é caracterizado pela possibilidade de se executar simultaneamente mais do que um kernel[32].

Para controlar a ordem e a dependência de comandos associados a cada *kernel*, a plataforma CUDA permite utilizar o que é chamado de *streams*. Esses comandos são representados pela chamada de algumas funções, denominadas funções assíncronas, e sua associação a um determinado *stream*. A ordem de execução dessas funções dentro de um *stream* é respeitada. Funções associadas a *streams* diferentes são independentes e podem ser executadas de forma concorrente. As chamadas de execução de cada *kernel* são implicitamente assíncronas. Todas as funções da API CUDA terminadas com o sufixo `Async` também são assíncronas.

Quando funções são chamadas e associadas a um *stream*, elas são enfileiradas numa fila de trabalho para serem executadas posteriormente pelo *runtime* CUDA. O fluxo de execução do código da aplicação no *host* não é bloqueado. Isso permite obter, além do paralelismo das tarefas na GPU, um paralelismo de tarefas na CPU.

Essa característica de execução dos *streams* permite realizar algumas tarefas importantes de forma simultânea, entre elas: (i) Execução de processamento na CPU e na GPU; (ii) Execução de processamento na CPU e cópia de dados entre a CPU e a GPU; (iii) Execução de processamento na GPU e cópia de dados entre a CPU e GPU e (iv) execução de vários *kernels* na GPU.

Considerando que uma das principais sobrecargas no processamento em GPU é o tempo relacionado a cópia de dados entre a CPU e GPU, a possibilidade de execução simultânea de processamento e cópia de dados torna-se bastante útil. Dependendo da relação que exista entre o tempo de cópia e o tempo de processamento, é possível compensar boa parte do tempo dispendido na cópia de dados.

Na Figura 4.7 é possível visualizar duas formas de execução de um *kernel*. Na parte superior está a forma normal, onde primeiro são transferidos todos os dados a serem processados para a GPU (CpG), chama-se o *kernel* para processá-los e transfere-se os dados processados da GPU para a CPU (GpC). Na parte inferior está a execução do mesmo *kernel*, com uso de três *streams*. Nesse caso, os dados são divididos e transferidos em partes. São feitas três chamadas

[32]Em GPUs com CC 6.0, é possível executar até 128 *kernels* simultaneamente.

do mesmo *kernel* para processar essas partes, cada uma em um *stream*. Pode-se perceber na linha de tempo como a cópia de dados da CPU para a GPU (CpG) e da GPU para a CPU (GpC) se sobrepõem à execução dos *kernels*. Consequentemente, obtém-se uma redução no tempo total de execução.

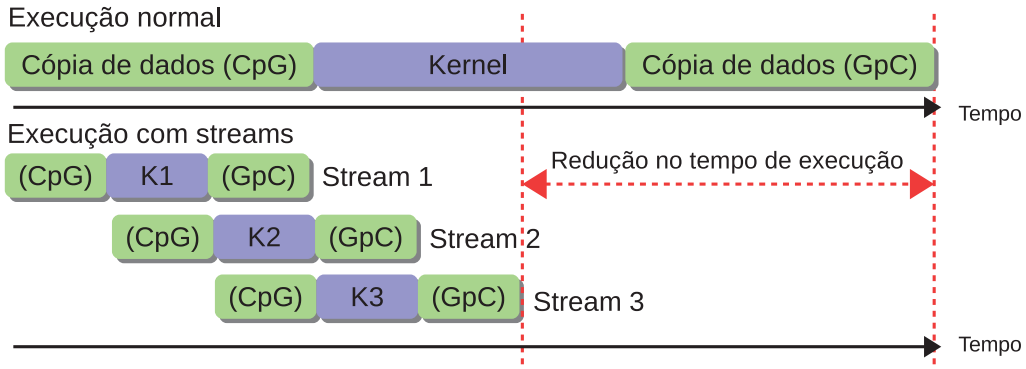


Figura 4.7: Redução no tempo total de execução com o uso de *streams*. Adaptado de [69].

A utilização das versões assíncronas para cópia de dados entre a CPU e GPU requerem que a memória alocada na CPU seja do tipo não-paginável. A API CUDA oferece funções específicas para realizar esse tipo de alocação: `cudaMallocHost()` e `cudaHostAlloc()`.

Também estão disponíveis funções que permitem criar pontos de sincronização ou consultar o estado de execução de um determinado *stream*. São as funções `cudaStreamSynchronize()` e `cudaStreamQuery()`, respectivamente. Esse controle também pode ser feito através da criação e monitoramento de eventos associados a um determinado *stream*. Funções como `cudaEventRecord()`, `cudaEventSynchronize()` e `cudaEventQuery()` podem ser utilizadas para essa finalidade.

4.5 Plataformas alternativas para processamento em GPUs

Devido ao fato da biblioteca WAESlib ser baseada na plataforma CUDA, este capítulo apresentou detalhes somente da arquitetura de GPUs NVIDIA. Porém, há GPUs de outros fabricantes, bem como formas alternativas de programação além da plataforma CUDA. Apenas como referência, algumas dessas alternativas serão apresentadas na sequência.

Outra fabricante conhecida de GPUs é a empresa *Advanced Micro Devices* (AMD). No mesmo segmento de GPUs discretas[33] e voltadas para o mercado profissional (*workstation*), seus modelos mais recentes[34] de GPUs podem ter até 4096 *stream processors* ou *shader processors* (análogos aos CUDA *cores* de GPUs NVIDIA). Também utilizam memórias do tipo HBM2, com largura de banda máxima teórica por volta dos 483 GB/s e capacidade de armazenamento de até 16 GB. Está baseada na microarquitetura e conjunto de instruções que a AMD denomina de quinta geração da *Graphics Core Next* (GCN) [77] e na arquitetura denominada Vega.

Também há o segmento de CPUs e GPUs integradas. Normalmente são utilizadas em dispositivos móveis, consomem menos energia e sua capacidade de computação, se comparada às

[33] O termo discreta refere-se ao fato de que a GPU e CPU são separadas e utilizam espaços de memória distintos, cuja comunicação acontece através de um barramento, geralmente o barramento PCIe. Quando a GPU e CPU compartilham o mesmo espaço de endereçamento, ocupando o mesmo chip, elas são denominadas integradas (*fused*). Nesse caso, a memória utilizada pela GPU é a mesma utilizada pela CPU [76].

[34] Valores referentes ao modelo Radeon Pro WX 9100 (https://pro.radeon.com/en-us/product/wx-series/radeon-pro-wx-9100/).

GPUs discretas, é significativamente menor. Porém, também podem ser utilizadas para realizar processamento paralelo em aplicações. Nesse segmento tanto a AMD quanto a Intel possuem produtos disponíveis.

No caso da Intel, alguns de seus processadores gráficos integrados a CPUs podem oferecer até 48 unidades de execução[35]. Cada unidade de execução pode executar até sete *threads*, totalizando 336 *threads*. A largura de banda disponível de acesso à memória é a mesma da memória de sistema, ficando por volta de 34 GB/s. As GPUs integradas que possuem essas características são as da nona geração de processadores gráficos da Intel e as CPUs possuem a microarquitetura denominada Kaby Lake [78][79].

A AMD também disponibiliza uma linha de CPUs e GPUs integradas denominada *Accelerated Processing Unit* (APU). Alguns modelos mais recentes[36] oferecem APUs com processador gráfico de até 512 *stream processors* e largura de banda de acesso à memória de sistema por volta de 38 GB/s. Fazem parte da sétima geração de APUs da AMD e são baseadas na microarquitetura de CPUs denominada *Excavator* [80].

Com relação à plataforma de programação, uma alternativa conhecida à plataforma CUDA é a *Open Computing Language* (OpenCL) [81][82]. OpenCL é um padrão aberto de programação paralela para diferentes tipos de processadores. Empresas como Intel, AMD, NVIDIA, ARM, entre outras, oferecem suporte à OpenCL na maioria de seus processadores. Assim, aplicações paralelizadas utilizando OpenCL podem ser executadas de forma transparente tanto em GPUs quanto CPUs.

O OpenCL tem suas particularidades, porém o modelo de programação é parecido ao modelo CUDA, utilizando nomes diferentes para estruturas semelhantes. Enquanto em CUDA utilizam-se termos como grade, bloco e *thread*, em OpenCL são utilizados termos como espaço de trabalho, grupo de trabalho e item de trabalho[37], respectivamente. Com relação aos componentes de hardware, NVIDIA/CUDA usam termos como *streamming multiprocessor*, CUDA *core* e *shared memory*, OpenCL usa termos como unidade de computação, elemento de processamento e memória local[38] [83].

Outro padrão que pode ser utilizado para paralelizar aplicações a fim de permitir que elas possam realizar processamento em CPUs e GPUs é chamado OpenACC [84][85]. Um dos objetivos desse padrão foi criar uma forma mais simplificada de programação paralela, a qual exigisse um menor esforço de programação. Seguindo o mesmo modelo do OpenMP, o OpenACC baseia-se no uso de diretivas de compilação (*pragmas*) que podem ser utilizadas pelo programador para identificar partes do código que devem ser executadas em paralelo. Também oferece uma API em tempo de execução que permite controlar alguns aspectos do ambiente e da execução paralela.

Em [76] há uma revisão interessante da literatura envolvendo computação heterogênea entre CPUs e GPUs. São relacionas diversas pesquisas envolvendo linguagens de programação, *frameworks* e outras ferramentas de desenvolvimento utilizadas para programação paralela. Também são abordadas pesquisas que envolvem CPUs e GPUs integradas. Em [86] é feito um estudo que compara quesitos como produtividade de programação, desempenho e consumo de energia para processamento paralelo utilizando CUDA, OpenCL, OpenACC e OpenMP.

[35] Valores referentes ao processador gráfico *Intel Iris Plus Graphics* 650. Esse modelo vem integrado ao processador Intel Core i7-7567U (https://ark.intel.com/products/97541/Intel-Core-i7-7567U-Processor-4M-Cache-up-to-4_00-GHz).

[36] Valores referentes à APU modelo FX 9830P (http://products.amd.com/en-us/search/APU/AMD-FX-Series-Processors/AMD-FX-Series-Processors-for-Laptops/7th-Gen-FX%E2%84%A2-9830P-APU/191).

[37] Tradução para os termos em inglês: *work-space*, *work-group* e *work-item*, respectivamente.

[38] Tradução para os termos em inglês: *compute unit*, *processing element* e *local memory*, respectivamente.

4.6 Considerações finais

Como parte da fundamentação teórica, neste capítulo foram abordados alguns assuntos relacionados ao processamento em GPU. Entre esses assuntos estão os relacionados tanto a questões de arquitetura de hardware quanto de software. Foi dado maior ênfase à solução oferecida pela fabricante NVIDIA pois a biblioteca WAESlib está baseada nela.

Foram apresentadas algumas características que diferenciam as GPUs de CPUs e uma breve descrição do seu processo evolutivo. Com relação à arquitetura, foram descritos os principais componentes das GPUs NVIDIA e suas funções (Seção 4.1). Na camada de software, foi apresentada a plataforma CUDA e sua extensão para a linguagem C. Foram discutidas as formas como as *threads* são organizadas e apresentado um exemplo típico de aplicação escrita em CUDA C (Seção 4.2). Também foram vistos os diferentes tipos de memória encontrados em GPUs e suas características (Seção 4.3). Foi apresentado o recurso de *streams* que permite paralelizar tarefas executadas na GPU (Seção 4.4).

Como forma de complementação, o capítulo é finalizado com uma breve descrição de plataformas alternativas para processamento em GPU. Na camada de hardware, são abordadas os processadores das fabricantes AMD e Intel. Na camada de software, são descritas as plataformas de programação OpenCL e OpenACC (Seção 4.5).

Tendo sido vistos cifradores simétricos de blocos, modos de operação, sistemas de armazenamento criptográficos e processamento em GPUs, os dois capítulos seguintes apresentam os detalhes referentes à implementação de um SAC atuando no espaço do usuário, utilizando o modo de operação CTR e o processamento das funções criptográficas em GPU.

Capítulo 5

Aplicação do modo CTR em um Sistema de Arquivos Criptográfico

A implementação do modo CTR em um SAC, além de permitir explorar os recursos do processamento paralelo, permite realizar a computação antecipada de máscaras de cifragem que podem ser empregadas para compensar a latência de processamento. Entretanto, questões de implementação do modo CTR podem anular esses benefícios. Um dos pontos importantes na implementação do CTR, no contexto de SACs, está relacionado à geração e armazenamento dos *nonces* que são utilizados nas operações de cifragem e decifragem de blocos.

Neste capítulo serão apresentadas técnicas que tratam exclusivamente as questões relacionadas ao gerenciamento dos *nonces*, bem como formas de implementá-las. Esta parte corresponde à primeira etapa do processo de preparação para que o SAC possa realizar o processamento das funções criptográficas em GPU de forma eficiente. A segunda etapa, que envolve o processamento das funções criptográficas em GPU e trata questões relacionadas ao gerenciamento dos contextos de cifragem, será abordada no capítulo seguinte.

Na Seção 5.1 são apresentadas as questões relativas à geração e armazenamento de *nonces* para o modo CTR; a Seção 5.2 apresenta uma proposta de implementação do modo CTR aplicada em um SAC; Na Seção 5.3 são apresentadas avaliações de desempenho da implementação proposta. A Seção 5.4 apresenta as considerações finais do capítulo.

5.1 Geração e armazenamento de *nonces*

Tratando a questão de geração de *nonces* e procurando atender ao requisito de unicidade do *nonce* exigido pelo modo CTR, este trabalho propõe uma forma determinística para sua geração [32]. A técnica se baseia num único contador global que é incrementado a cada escrita e reescrita de bloco. Os números gerados por esse contador são utilizados como *nonce* nas funções de cifragem e decifragem. Considerando-se, por exemplo, a utilização do cifrador AES, cujos blocos possuem 128 bits, o *nonce* também precisa ter esse tamanho. Portanto, pode-se utilizar um contador de 128 bits.

Em implementações como a da biblioteca OpenSSL, os bits menos significativos do *nonce* são utilizados como um contador interno incrementado a cada 16 bytes de dados cifrados. A quantidade de bits necessários para esse contador é dada pela fórmula $log_2(x/16)$, onde x corresponde a quantidade de bytes a serem cifrados com esse *nonce*. Como os *nonces* devem ser únicos, a quantidade de blocos que podem ser escritos ou reescritos é dada pelo valor 2^{128-x}, onde x é a quantidade de bits reservados. A Figura 5.1 ilustra o formato do contador geral e os bits menos significativos reservados para o contador interno do modo CTR.

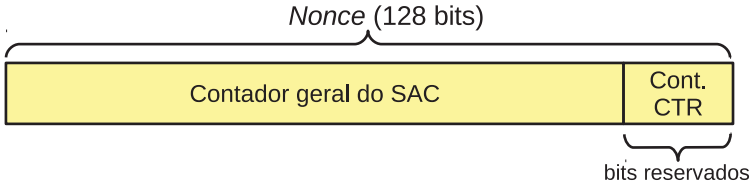


Figura 5.1: Formação de um *nonce* a partir do contador geral do SAC.

Em relação ao armazenamento de *nonces*, quando um bloco é escrito, o *nonce* utilizado no seu processo de cifragem é obtido do contador geral do SAC. Num processo futuro de leitura do mesmo bloco, para que seja possível decifrar seu conteúdo, é necessário que o mesmo *nonce* seja passado como parâmetro às funções de decifragem. Portanto, é necessário que ele seja armazenado para futura recuperação.

Uma ideia simples seria armazenar o *nonce* individualmente antes de cada bloco. Porém, essa abordagem é inadequada à leitura aleatória pois dificulta a leitura prévia dos *nonces* com o objetivo de acionar a geração antecipada das máscaras de cifragem. Além disso, como os *nonces* possuem 128 bits, isso resulta numa estrutura de armazenamento não alinhada com o tamanho comum de páginas de memória do SO e blocos de dados dos SABs, prejudicando o desempenho.

Uma forma de contornar esse problema é armazená-los separadamente do arquivo cifrado. Assim, eles ocupam regiões contíguas em disco, agilizando os processos de leitura e escrita. Além disso, como cada *nonce* possui apenas 16 bytes, realizar seu acesso de maneira individual não é eficiente. Além de armazená-los em arquivos separados, também é interessante que eles sejam lidos e escritos de forma agrupada.

Contudo, o armazenamento agrupado e em arquivos separados não é a solução ideal para todos os casos. Em um cenário que trabalhe com arquivos muito pequenos, contendo poucos blocos de dados, ler e escrever um grupo inteiro de *nonces* pode ocasionar desperdício de memória e espaço em disco. O ideal é armazenar os *nonces* iniciais de todos os arquivos em um local de armazenamento único. Somente quando um arquivo ocupe uma quantidade maior de blocos, seus demais *nonces* começam a ser armazenados em um arquivo separado e exclusivo.

Para lidar com a situação de armazenamento dos *nonces* iniciais dos arquivos, este trabalho propõe uma solução inspirada no sistema de armazenamento de *inodes* do Unix. Essa estrutura foi chamada de *nonce node* (*nnode*). Existe um único arquivo responsável por armazenar os *nnodes* de todos os arquivos armazenados no SAC. A estrutura geral desse arquivo pode ser vista na Figura 5.2 (a).

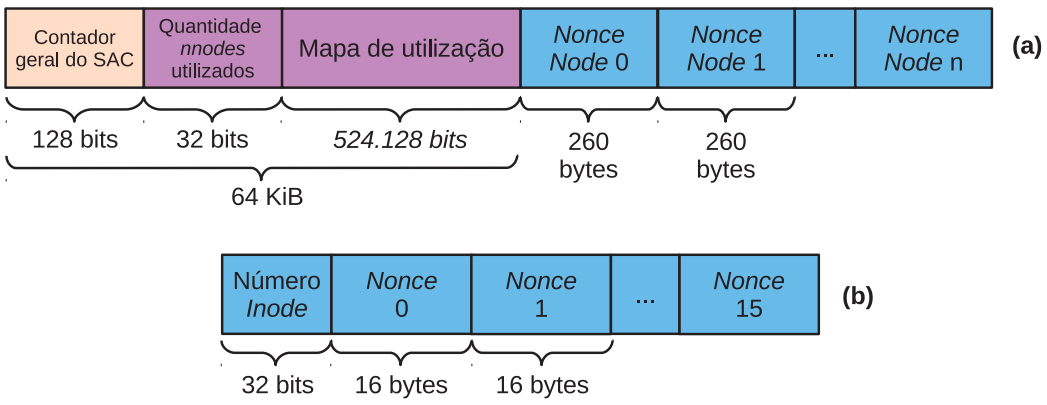


Figura 5.2: Formato geral do arquivo de *nnodes* (a). Detalhamento de um *nnode* (b).

No início do arquivo é armazenado o contador geral do SAC. Na sequência há um contador que controla a quantidade de *nnodes* utilizados, bem como um mapa de bits que tem por finalidade controlar a alocação de *nnodes*. Os *nnodes* em si vêm na sequência. Cada arquivo armazendo no SAC possui um *nnode* alocado para ele e armazenado nessa estrutura. Na expansão de um *nnode*, ilustrada na Figura 5.2 (b), pode-se ver as informações contidas nele. São o número de *inode* do arquivo (utilizado para indexar o arquivo ao *nnode*), seguido dos 16 primeiros *nonces* do arquivo. O tamanho do arquivo de *nnodes* cresce à medida que novos arquivos são armazenados no SAC.

Considerando-se que os *nnodes* servem para armazenar os 16 primeiros *nonces* de um arquivo e caso o SAC use blocos de 4 KiB, isto significa que para arquivos de até 16 blocos, ou seja, 64 KiB, apenas a estrutura de um *nnode* é suficiente para guardar seus *nonces*. Essa forma de armazenamento ajuda a otimizar o acesso aos *nonces* de arquivos pequenos pois permite que eles sejam armazenados em regiões próximas no disco. Além disso, também contribui para não desperdiçar espaço em disco alocando estruturas maiores para armazenamento de um grupo inteiro de *nonces*.

Caso um arquivo cresça além dos 16 blocos, os demais *nonces* começam a ser armazenados em um arquivo separado e exclusivo para cada arquivo armazenado no SAC. O formato desse arquivo pode ser visto na Figura 5.3. Os *nonces* são armazenados em grupos de 256 *nonces* para coincidir com o tamanho de uma página de memória de 4 KiB. A medida que o arquivo cresce, novos grupos de *nonces* são criados e armazenados no arquivo.

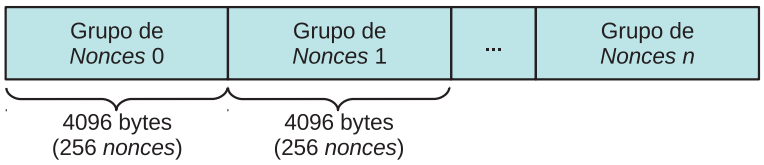


Figura 5.3: Formato do arquivo exclusivo de *nonces* para arquivos com mais de 16 blocos.

Os arquivos de *nonces* são armazenados num conjunto específico de diretórios, os quais não aparecem dentro do SAC montado. O nome de cada arquivo de *nonce* é derivado da aplicação de uma função de *hash* criptográfico (HMAC) sobre o *inode* do arquivo sendo cifrado e ao qual esse arquivo de *nonces* corresponde. De acordo com o valor em hexadecimal dos dois primeiros bytes do nome derivado, ele é armazenado em um desses diretórios.

5.2 Implementação do modo CTR

Procurando evitar o retrabalho de criar um novo SAC com o objetivo de validar as técnicas anteriormente descritas, optou-se por utilizar um SAC já existente, adaptando-o conforme necessário. Para tanto foi escolhido o EncFS. Além de ser um SAC conhecido, o fato de ser baseado em FUSE simplifica e agiliza os processos de desenvolvimento e testes. Visando entender melhor as alterações feitas, a subseção seguinte faz uma breve apresentação de seu funcionamento em sua implementação original.

5.2.1 EncFS

O EncFS foi brevemente apresentado na Seção 3.2. Nesta seção serão apresentados mais detalhes sobre como são cifrados os blocos e principalmente quais classes estão envolvidas no tratamento das requisições de leitura e escrita. A apresentação das classes é essencial para

posterior entendimento de como elas foram alteradas no processo de implementação do modo CTR.

O diagrama de sequência ilustrado na Figura 5.4[1] apresenta as principais classes envolvidas no tratamento de uma requisição `write()` proveniente do módulo FUSE. Foi escolhido apresentar esse tipo de requisição pois ela é mais complexa, envolvendo posteriormente a geração de *nonces*, a qual será explicada quando forem descritas as alterações realizadas.

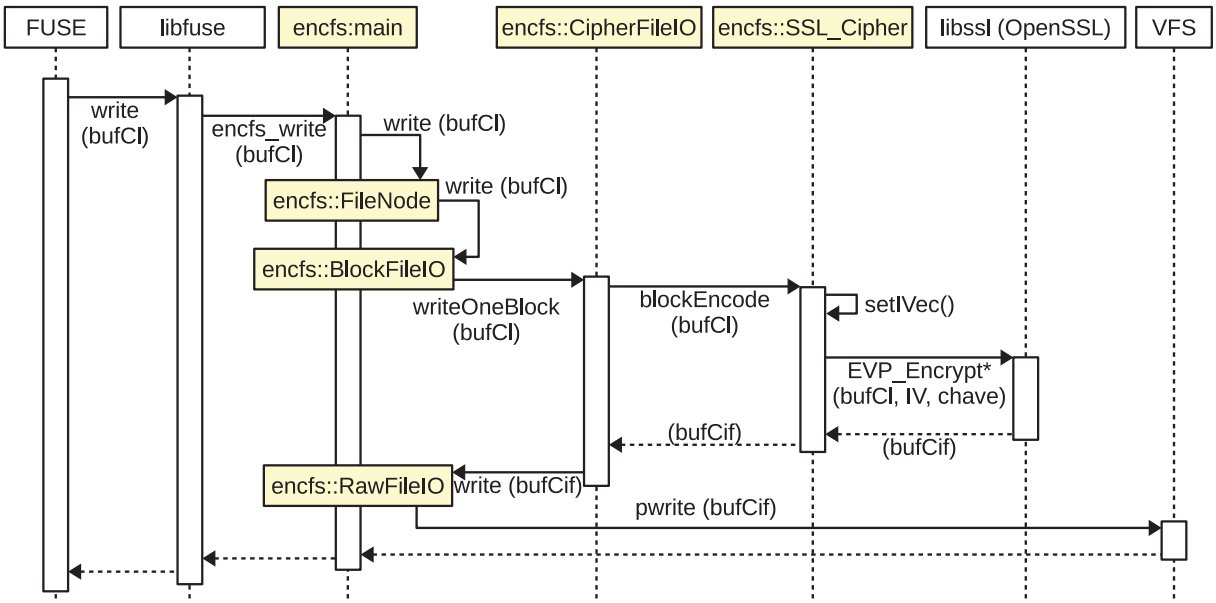


Figura 5.4: Processamento de uma requisição `write()` na implementação original do EncFS.

No tratamento da requisição, a *thread* acionada pela libfuse chama a função `encfs_write()` do EncFS. Após a recuperação do contexto de execução, armazenado em um objeto da classe `EncFS_Context`, é possível dar continuidade ao processo. Um objeto da classe `FileNode` representa o arquivo sendo escrito. Ele é instanciado quando um arquivo novo é criado ou um arquivo existente é aberto. A função `encfs_write()` chama o método `write()` do objeto `FileNode`. O objeto `FileNode` chama o método `write()` da classe `BlockFileIO`. O processo de entrada e saída intermediário é controlado pelo objeto dessa classe. A lógica que controla como as requisições de leitura e escrita são tratadas, ou seja, quais blocos são afetados e como eles são acessados, está contida nele. Após determinar como os blocos serão acessados, o método `writeOneBlock()` da classe `CipherFileIO` é chamado.

O controle da cifragem dos dados começa na classe `CipherFileIO`. Até esse ponto, nas chamadas dos métodos, o parâmetro `buffer(bufCl)`, contendo os dados a serem escritos, encontra-se em texto claro. Objetos da classe `CipherFileIO` possuem atributos que referenciam objetos da classe `SSLKey` e `SSL_Cipher`, os quais são criados na inicialização do EncFS. A chave de cifragem do volume fica no objeto `SSLKey`. Tanto o objeto `SSLKey`, quanto o `SSL_Cipher`, possuem referências para o contexto de cifragem do OpenSSL. Considerando que o *buffer* a ser escrito no arquivo corresponda exatamente ao tamanho do bloco, o método `blockEncode()` da classe `SSL_Cipher` é invocado.

Na classe `SSL_Cipher` há quatro métodos diretamente relacionados às operações de cifragem e decifragem: `blockEncode()`, `blockDecode()`, `streamEncode()` e

[1]Procurando facilitar a visualização, esse diagrama ilustra somente as classes e interações mais relevantes para compreensão do tratamento da requisição. O diagrama também foi levemente adaptado para melhor utilização do espaço disponível.

`streamDecode()`. Leitura e escrita de blocos inteiros usam os dois primeiros, blocos parciais os dois últimos.

A geração dos IVs é feita no método `setIvec()`. São usados e encadeados três tipos diferentes de IVs. O primeiro é o IV contido na chave do volume (IVV). O segundo é o IV do arquivo (IVA) e o terceiro são os IVs para cifragem dos blocos dos arquivos (IVB). O IVV está armazenado nos bytes finais da chave de volume (VK). O IVA é gerado aleatoriamente. O Quadro 5.1 descreve como o IVB é gerado. A notação `HMAC_CTX()` representa a geração de um *hash* criptográfico que utiliza os parâmetros mostrados e o qual é produzido através da chamada a algumas funções da biblioteca OpenSSL. O símbolo "‖" significa concatenação, o símbolo "⊕" representa a operação de `XOR` e *NumBloco* corresponde ao número do bloco dentro do arquivo. O IVA é escrito de forma cifrada no cabeçalho do arquivo.

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

Quadro 5.1: Função de geração de IVs para cifragem e decifragem de blocos.

Com o *buffer* em texto claro, o IVB e a chave de cifragem do volume, o objeto da classe `SSL_Cipher` pode acionar o contexto de cifragem do OpenSSL para cifrar o *buffer* (chamada às funções `EVP_Encrypt*`). Quando o processo de cifragem termina, o *buffer*, inicialmente passado às funções de cifragens, conterá o bloco cifrado (`bufCif`) e ocorre o retorno ao objeto da classe `CipherFileIO`. O método `write()` do objeto da classe `RawFileIO` é chamado. Finalmente é chamada a função `pwrite()` para escrever os dados cifrados no SAB. A partir desse ponto a escrita segue o seu curso normal através do VFS. O formato final de um arquivo EncFS pode ser visto na Figura 5.5.

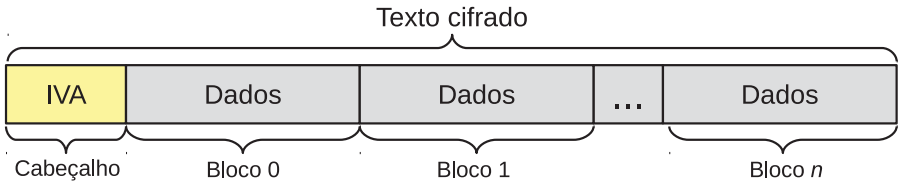


Figura 5.5: Formato de um arquivo armazenado no SAC EncFS.

5.2.2 Alterações no EncFS

Em seu modo padrão de operação e utilizando o cifrador AES, a definição do contexto de cifragem para blocos no EncFS utiliza as funções `EVP_aes_*_cbc()`[2] da biblioteca OpenSSL. Um novo cifrador foi criado, chamado AESCTR, o qual utiliza em sua definição do contexto de cifragem para blocos as funções `EVP_aes_*_ctr()`. Dessa forma, as cifragens de blocos utilizando esse cifrador dentro do EncFS passam a utilizar o modo de operação CTR implementado na biblioteca OpenSSL.

Um dos primeiros métodos executados na inicialização da nova versão do EncFS é o `encfs::initializeFSNNodeFiles()`. Quando o SAC é criado, ele é responsável por criar e inicializar o arquivo principal de armazenamento dos *nnodes*. O método `encfs::loadFSNNodeFile()` tem como principal objetivo percorrer o mapa de ocupação,

[2]Onde * corresponde ao tamanho em bits da chave seleciona quando da criação do SAC.

criando duas estruturas importantes: uma lista de *nnodes* livres (`listFreeNNodes`) e outra lista (`nNodeMap`) onde são mapeados os *nnodes* com seus respectivos *inodes*. A primeira lista agiliza a localização de *nnodes* livres que podem ser alocados para arquivos novos. A segunda agiliza o acesso às estruturas em memória que armazenam os *nnodes* dos arquivos. Ao realizar esse mapeamento, todos os *nnodes* dos arquivos existentes são carregados em memória. Esse carregamento prévio ajuda de forma significativa a reduzir o impacto no desempenho do SAC ao trabalhar com arquivos de tamanho pequeno.

Também é mapeado em memória, através da função `mmap()`, a região inicial do arquivo de *nnodes* onde estão armazenados o contador geral do SAC, o contador que controla a quantidade de *nnodes* utilizados e o mapa de ocupação. São estruturas acessadas com frequência e cuja sincroniza de dados entre memória e disco é muito importante. Esse mapeamento é utilizado para agilizar o acesso a elas, ao mesmo tempo em que oferece um nível maior de integridade ao SAC, pois o próprio SO se encarrega de sincronizar constantemente os dados entre memória e disco.

A alocação de um *nnode* só ocorre quando dados começam a ser escritos no arquivo, ou seja, não são alocados *nnodes* para arquivos com tamanho igual a zero. Quando essa escrita inicia, é chamado o método `EncFS_Context::getAllocNNode()`. Ele procura na lista `nNodeMap` se já existe um *nnode* mapeado para o *inode* do arquivo que está sendo escrito. Caso exista, retorna o *nnode* encontrado. Essa situação só ocorre com arquivos que já sofreram processos prévios de escrita e que, portanto, já possuem um *nnode* alocado. Caso não exista, o método retira o primeiro *nnode* livre da lista `listFreeNNodes` e o devolve no retorno do método. O bit correspondente a esse novo *nnode* alocado é marcado como sendo utilizado no mapa de ocupação e o mapeamento desse *nnode* é acrescentado à lista `nNodeMap`. Há um método chamado `EncFS_Context::releaseNNode()` com funcionalidades semelhantes e inversas, utilizado na remoção de arquivos.

No tratamento das requisições de abertura de arquivo, o método `FileNode::loadNNode()` é responsável por chamar o método `EncFS_Context::getAllocNNode()` a fim de obter acesso ao seu *nnode*. Caso o arquivo contenha mais do que 16 blocos, deduz-se que ele também possui um arquivo exclusivo onde estão armazenados os *nonces* dos demais blocos. Portanto, é chamado o método `FileNode::loadNonceGroups()` que se encarrega de carregar em memória os demais *nonces*. Uma referência a cada grupo de *nonce* carregado é mantida num vetor chamado `nonceGroups`. O carregamento prévio do *nnode* do arquivo durante a inicialização do SAC, e de seus grupos de *nonces* na abertura, garantem que todos os *nonces* necessários para os processos de decifragem estejam diretamente acessíveis em memória. Essas duas abordagens são essenciais para um bom desempenho das funções de leitura. Além disso, tem uma baixa sobrecarga de memória. Considerando que o tamanho do bloco escolhido seja de 4 KiB e que cada bloco precisa de um *nonce* de 16 bytes, a sobrecarga é de aproximadamente 0,004%.

As requisições de leitura e escrita possuem tratamentos semelhantes. Por ser mais complexa, optou-se por detalhar como foi tratada a questão dos *nonces* nos processos de escrita. Ela está ilustrada no diagrama de sequência da Figura 5.6 e descrito na sequência.

Comparando com o diagrama ilustrado na Figura 5.4 da Seção 5.2.1, após chegar no método `FileNode::write()`, são chamados os métodos `FileNode::loadNNode()` e `FileNode::loadNonceGroups()`. Na escrita, caso o arquivo ainda esteja zerado, será alocado um novo *nnode* para ele através do chamado ao método `EncFS_Context::getAllocNNode()`. Caso a escrita afete blocos acima dos 16 primeiros, o método `FileNode::loadNonceGroups()` também se encarrega de criar o arquivo

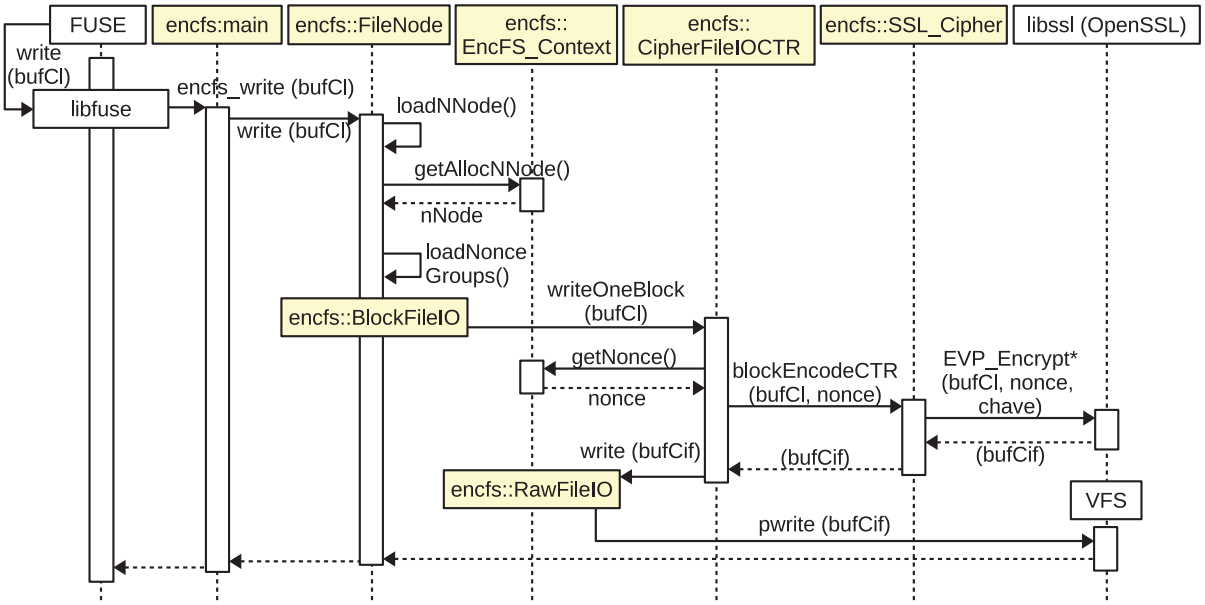


Figura 5.6: Processamento de uma requisição `write()` na implementação do modo CTR.

adicional de armazenamento de *nonces*. Além disso, aloca espaço em memória para armazenar os novos *nonces* que serão gerados e utilizados no processo de cifragem dos blocos.

Nas situações em que o arquivo já possua dados gravados, os métodos `FileNode::loadNNode()` e `FileNode::loadNonceGroups()`, mesmo no tratamento das requisições de escrita, também fazem a leitura de *nonces* preexistentes pois os mesmos são necessários nos casos da escrita de blocos parciais. Esta situação requer primeiro a decifragem do bloco, inclusão dos dados a serem escritos, recifragem do bloco e por fim sua reescrita em disco.

Foi implementada uma nova classe chamada `encfs::CipherFileIOCTR` baseada na classe `encfs::CipherFileIO`, cujos métodos mais alterados foram o `CipherFileIOCTR::writeOneBlock()` e `CipherFileIOCTR::readOneBlock()`. No caso da escrita, o método chamado é o `CipherFileIOCTR::writeOneBlock()`. Baseado no *offset* da operação e o tamanho do bloco utilizado pelo SAC, deduz-se qual bloco será afetado na escrita. Dos blocos de zero até 15, os *nonces* utilizados serão armazenados no *nnode*, acima de 15 no vetor `nonceGroups`. O número do bloco serve como índice para acesso direto aos *nonces* armazenados tanto no *nnode* quanto no vetor `nonceGroups`.

Antes de ser chamado o método `SSL_Cipher::blockEncodeCTR()`, é preciso gerar o *nonce* que será utilizado no processo de cifragem do bloco. Essa é a finalidade do método `EncFS_Context::getNonce()`. Ele faz o incremento do contador geral do SAC e devolve o valor que será utilizado como *nonce*. De acordo com o número do bloco para o qual está sendo gerado o *nonce*, é configurada uma *flag* para marcar o grupo de *nonce* específico ou *nnode* como sendo sujo. Esse processo é útil em outro método que controla a escrita em disco dos *nonces* marcados como sujos que estão em memória.

Foram criados dois métodos novos na classe `SSL_Cipher`: `blockEncondeCTR()` e `blockDecodeCTR()`. No caso da escrita, o primeiro é chamado. É neste método onde

efetivamente são chamadas as funções de cifragem do bloco (`EVP_Encrypt*`[3]). Nele acessa-se o contexto de cifragem da biblioteca OpenSSL[4] e passam-se os dados referentes ao conteúdo do bloco a ser cifrado, a chave e o *nonce*. Após o término da execução das funções, os dados referentes ao bloco contido no *buffer* estarão cifrados. O restante do processo ocorre de forma idêntica ao modo original de funcionamento do EncFS, onde o bloco cifrado passa pela classe `RawFileIO` que o encaminha ao VFS para escrita no SAB.

Considerando que tanto *nnodes* e `nonceGroups` são mantidos em memória, é necessário garantir sua persistência em meio persistente. Com esta finalidade foram implementadas duas políticas de sincronização.

A primeira trata da sincronia dos arquivos exclusivos de *nonces*. Foi implementado um método denominado `FileNode::_flushNonces()` que é chamado após o tratamento normal de operações de `sync` sobre o arquivo de dados. O objetivo desse método é percorrer o vetor `nonceGroups` procurando por grupos de *nonces* que estejam com a *flag* de sujo marcada. Os grupos encontrados são então escritos em disco e a *flag* removida. Após finalizado o processo, uma operação de `sync` é chamada sobre o arquivo de *nonces* para forçar sua escrita em disco. O mesmo método também é chamado no fechamento do arquivo.

A segunda está relacionada com a sincronia dos *nnodes*, sendo mantida em nível do SAC e controlada por uma *thread* que executa o método `EncFS_Context::flushNNodeData()` a cada 30 segundos. Seu objetivo é semelhante ao do método `FileNode::_flushNonces()`, porém percorrendo uma lista de *nnodes* marcados como sujos (`listDirtyNNodes`), forçando sua escrita em disco e removendo-os desta lista. Ao final, é forçada a sincronização do arquivo de *nnodes*. O mesmo método também é chamado na desmontagem do SAC.

O tratamento das operações de leitura são semelhantes, porém mais simples pois não necessitam lidar com a geração de novos *nonces*. Por esse motivo, optou-se por não ilustrar e detalhar seu funcionamento. Conforme já descrito, os *nnodes* dos arquivos já são carregados em memória na inicialização do SAC e os *nonces* armazenados em arquivos exclusivos são carregados quando o arquivo é aberto. Por isso, no método `CipherFileIOCTR::readOneBlock()`, basta acessar as estruturas de armazenamento dos *nonces* em memória para ter acesso ao *nonce* específico a ser utilizado na decifragem do bloco sendo acessado.

5.3 Avaliações de desempenho

Nas subseções seguintes são apresentadas análises de desempenho do tipo *microbenchmark* e *macrobenchmark*, comparando o desempenho do EncFS operando no seu modo padrão, o CBC, e no modo CTR implementado. Na primeira foram analisadas as operações de leitura e escrita sequenciais e aleatórias visando avaliar o desempenho em termos de vazão. Também foram avaliados possíveis ganhos cumulativos em um cenário com acesso realizado por múltiplos processos, bem como com múltiplos contextos de cifragem.

Nas análises de *macrobenchmark* foram avaliadas predominantemente operações de leitura e escrita sequenciais aplicadas sobre cargas de trabalhando contendo arquivos de tamanhos e quantidades variadas. Neste cenário foi analisado o aumento ou redução no tempo envolvido na realização das cópias dos arquivos das diferentes cargas. Também foram feitas análises de latência sobre estas operações.

[3] São utilizadas as funções `EVP_EncryptInit_ex()` para inicializar o contexto com o *nonce* a ser utilizado; `EVP_EncryptUpdate()` para passar os dados a serem cifrados e `EVP_EncryptFinal_ex()` para finalizar o processo de cifragem e obter acesso ao *buffer* contendo os dados já cifrados.

[4] Cuja configuração para usar o modo CTR é feita na inicialização do SAC.

5.3.1 Vazão usando *microbenchmark*

Para esta análise foi utilizada a ferramenta `fio` versão 3.2-72, configurada para operar sobre um único arquivo com 16 GiB (dobro da memória RAM). Foi utilizada a opção `refill_buffers` para que os dados escritos no arquivo fossem totalmente aleatórios e a opção `allrandrepeat` para que todas as funções baseadas em aleatoriedade apresentassem o mesmo comportamento entre diferentes execuções da ferramenta. A ferramenta foi configurada para enviar uma requisição de sincronia a cada 16 MiB de dados escritos.

Foram medidos níveis de vazão com operações variando o tamanho das requisições de 1 KiB até 512 KiB[5]. Foram medidos quatro tipos diferentes de operação: escrita sequencial, escrita aleatória, leitura sequencial e leitura aleatória. Cada medição foi executada por um período de 60 segundos[6] e repetida 10 vezes, sendo calculada a média aritmética simples dos resultados. Entre cada repetição foi executado um comando para realizar o descarte das páginas de memória em *cache*. Foi utilizada uma partição específica para a criação do SAB. Entre os diferentes cenários avaliados, o SAB era desmontado, recriado e remontado. O SAB utilizado foi o ext4 com as configurações padrões.

Os testes foram realizados utilizando-se o Linux com *kernel* versão 4.4.0, rodando em processador Intel Core i7 920 a 2,67 GHz (frequência fixa e com quatro núcleos físicos de processamento), 8 GiB de memória RAM DDR3 atuando em *dual channel* e disco Western Digital modelo WD5000AZLX-0 com taxa de vazão sustentada teórica máxima por volta de 146.484 KiB/s (aproximadamente 143,05 MiB/s). A versão utilizada da biblioteca libfuse foi a 2.9.2 e da biblioteca OpenSSL 1.0.1f. Procurando reduzir variações na capacidade de processamento, foram desabilitadas as funcionalidades específicas de processadores Intel denominadas *Hyper-Threading* e *Turbo Boost*.

O experimento foi projetado de forma a diminuir ao máximo variáveis fora de controle, as quais pudessem afetar o resultado. Os principais objetivos foram: (i) Procurar reduzir ao máximo a atuação do mecanismo de *cache* de páginas, entretanto, sem anulá-lo; (ii) Ler e escrever dados aleatórios reais e com os mesmos padrões de acesso aleatório; (iii) Ler e escrever dados sempre nas mesmas regiões em disco.

A Figura 5.7 (a) e (b), apresenta as medições de vazão para as operações de escrita e leitura sequenciais. Por considerar o ganho ou perda de vazão o dado mais relevante, ele é apresentado no eixo *y* principal e representado no gráfico pela linha. São apresentados em forma de porcentagem, sendo que valores positivos indicam ganho de vazão do modo CTR em relação ao CBC e valores negativos indicam perda. No eixo secundário *y* são apresentados os níveis de vazão alcançados em ambos os modos e representados pelas barras verticais.

As Tabelas 5.1 e 5.2 apresentam os valores referentes aos dados graficados. Na primeira coluna estão os tamanhos das requisições. As quatro colunas seguintes apresentam os valores de vazão mínima, máxima, desvio padrão e média para o modo CBC. As quatro colunas seguintes contém os valores de vazão alcançados pelo modo CTR. A última coluna contém o percentual de ganho ou perda na vazão comparando-se a vazão alcançada no modo CTR em relação ao modo CBC. Os valores abaixo de zero foram coloridos em vermelho, indicando perda de desempenho e, acima de zero em verde, indicando ganho de desempenho.

[5] As requisições provenientes da libfuse, por padrão, estão limitadas a 32 páginas de memória (128 KiB). Portanto, não há necessidade de medir requisições com tamanhos significativamente maiores do que este valor.

[6] A escolha de um tempo mais curto teve por objetivo evitar que em requisições de tamanho maior, principalmente no caso da leitura, ocorresse a situação em que todo o arquivo é lido. Caso isto ocorra e o tempo ainda não tenha terminado, a ferramenta começa a acessar partes do arquivo que já foram acessadas previamente, consequentemente obtendo dados armazenados em memória *cache* e distorcendo os resultados.

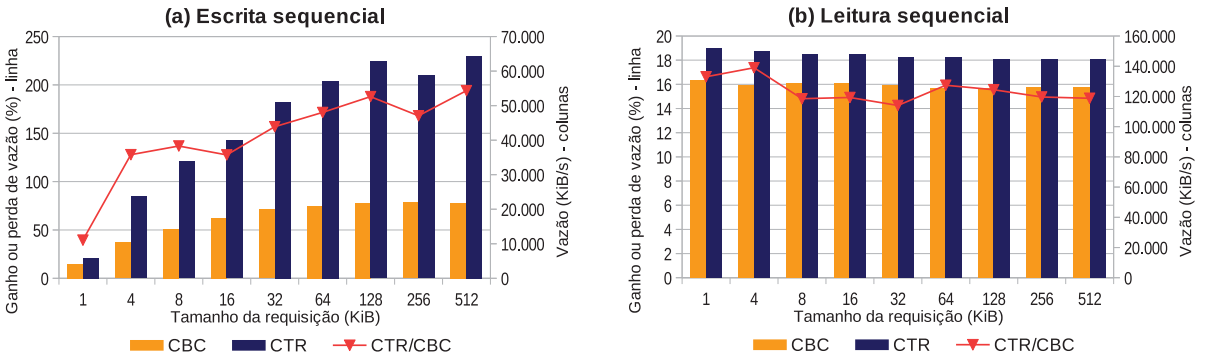


Figura 5.7: Níveis de vazão em escrita sequencial e leitura sequencial, com 1 processo.

Tabela 5.1: Níveis de vazão para escrita sequencial com 1 processo.

Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
	CBC				CTR				
	Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	4.148	4.316	60,61	4.229,2	5.699	6.154	149,4	5.898,2	39,46
4	10.302	10.610	106,39	10.403,6	22.815	24.550	579,52	23.705,6	127,86
8	13.910	14.845	357,46	14.296,4	32.250	35.289	1.145,11	33.858,8	136,83
16	17.237	17.890	249,58	17.524,6	36.421	44.206	3.066,67	39.897	127,66
32	19.227	20.821	585,79	19.873,4	46.824	56.099	3.422,02	51.027	156,76
64	20.690	21.364	239,32	21.024,2	53.757	61.128	2.745,88	57.087,6	171,53
128	21.217	22.698	501,5	21.817,2	58.042	65.450	2.701,85	62.816,6	187,92
256	21.240	22.541	453,96	21.910	57.286	61.646	1.533,92	58.736,2	168,08
512	21.455	22.456	362,59	21.870,6	58.956	66.628	2.894,75	64.314,4	194,07
Média	**16.602,89**	**17.504,56**	**324,13**	**16.994,36**	**41.338,89**	**46.794,44**	**2.026,57**	**44.149,04**	159,79

Tabela 5.2: Níveis de vazão para leitura sequencial com 1 processo.

Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
	CBC				CTR				
	Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	129.980	131.387	497,23	130.540,8	149.728	153.948	1.534,16	152.247,4	16,63
4	127.279	128.250	336,04	127.611,6	148.305	150.415	773,36	149.786,4	17,38
8	128.300	129.364	409,81	128.952,8	146.612	149.489	1.029,05	148.050,4	14,81
16	127.800	129.568	638,87	128.666	146.777	149.967	1.122,31	147.851,2	14,91
32	126.176	128.527	897,79	127.577,4	144.384	147.040	882,08	145.763,4	14,25
64	124.403	126.835	818,63	125.730,8	145.151	146.580	516,67	145.753,6	15,93
128	124.664	126.030	571,29	125.386,6	144.035	145.732	709,71	144.891,6	15,56
256	124.640	128.602	1.408,64	126.065,2	143.499	146.285	939,43	144.919,6	14,96
512	124.830	128.628	1.359,79	126.011	142.593	146.580	1.497,97	144.724,4	14,85
Média	**126.452,44**	**128.576,78**	**770,9**	**127.393,58**	**145.676**	**148.448,44**	**1.000,53**	**147.109,78**	15,48

Na operação de escrita sequencial há ganhos de desempenho em todos os tamanhos de requisição, mais significativamente com requisições acima de 1 KiB. Na média geral o ganho situa-se em 159,79%. Neste cenário já se pode perceber o quanto a capacidade de paralelização da cifragem no modo CTR pode contribuir para uma melhora significativa no desempenho da escrita, enquanto a serialização do modo CBC se torna um fator limitante.

Na leitura sequencial os ganhos são mais modestos, ficando em média por volta de 15,48%. O processo de decifragem do modo CBC também pode ser paralelizado, portanto dificilmente se conseguiria ganhos mais significativos explorando apenas essa característica. Neste cenário, o mecanismo de leitura antecipada de dados em disco realizado pelo SO vai

disponibilizando os dados antecipadamente em memória, consequentemente não há grandes variações de vazão entre os diferentes tamanhos de requisição.

A Figura 5.8 (a) e (b) apresenta os gráficos referentes a escrita e leitura aleatórias, seguidos pelas Tabelas 5.3 e 5.4 contendo os valores correspondentes.

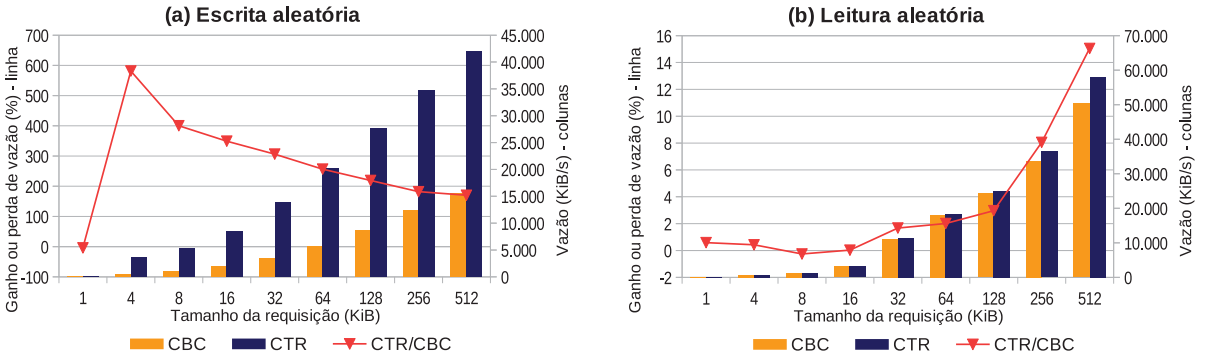


Figura 5.8: Níveis de vazão em escrita aleatória e leitura aleatória, com 1 processo.

Tabela 5.3: Níveis de vazão para escrita aleatória com 1 processo.

Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
	CBC				CTR				
	Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	146	151	1,96	149,6	135	151	6,09	143,6	-4,01
4	525	549	8,7	536,2	3.617	3.670	19,93	3.653,6	581,39
8	1.050	1.060	4,12	1.056,8	5.233	5.325	32,85	5.288,2	400,4
16	1.870	1.880	3,44	1.875,4	8.354	8.528	60,96	8.422,8	349,12
32	3.422	3.446	7,76	3.432,4	13.594	14.244	232,27	13.948	306,36
64	5.624	5.732	42,7	5.680,6	20.092	20.434	144,11	20.263	256,71
128	8.665	8.713	15,77	8.685,6	27.493	28.137	223,17	27.725,4	219,21
256	12.171	12.383	89,96	12.297,2	33.628	36.291	915,01	34.675,8	181,98
512	15.231	15.656	155,77	15.537,6	40.589	44.890	1.545,41	41.966,4	170,1
Média	**5.411,56**	**5.507,78**	**36,69**	**5.472,38**	**16.970,56**	**17.963,33**	**353,31**	**17.342,98**	216,92

Tabela 5.4: Níveis de vazão para leitura aleatória com 1 processo.

Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
	CBC				CTR				
	Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	170	170	0	170	171	171	0	171	0,59
4	666	666	0	666	668	669	0,4	668,8	0,42
8	1.303	1.307	1,26	1.305	1.300	1.304	1,36	1.301,6	-0,26
16	3.113	3.130	5,95	3.121,2	3.118	3.125	2,61	3.122	0,03
32	11.140	11.171	12,52	11.158,4	11.303	11.400	40,24	11.344,8	1,67
64	17.930	18.027	34,16	17.983	18.280	18.418	52,82	18.344,6	2,01
128	24.187	24.345	57,87	24.254,2	24.705	25.192	165,44	24.972,8	2,96
256	33.639	33.804	62,12	33.708,8	36.297	36.512	77,37	36.422,8	8,05
512	50.031	50.763	237,1	50.387,2	57.858	58.104	97,99	57.970,6	15,05
Média	**15.797,67**	**15.931,44**	**45,66**	**15.861,53**	**17.077,78**	**17.210,56**	**48,69**	**17.146,56**	8,1

Em requisições com tamanho de 1 KiB praticamente não há ganho ou perda na leitura aleatória e uma pequena perda de 4% na escrita aleatória. Neste caso em específico, a escrita de 1 KiB resulta na escrita parcial de blocos, pois o SAC foi configurado para utilizar blocos de 4 KiB. Consequentemente ocorre a necessidade da leitura prévia de um bloco, escrita de 1

KiB e recifragem do bloco novamente. Isto acaba anulando ganhos mais significativos que a escrita com requisições maiores oferece. Em compensação, os demais tamanhos de requisição apresentam ganhos bastante significativos em relação ao modo CBC, ficando em média por volta de 216,92%.

Há outro fator que também amplia os ganhos alcançados pela utilização do modo CTR. O armazenamento dos *nonces* iniciais nos *nnodes* e os demais em arquivos exclusivos fazem com que não seja preciso guardar dados referentes a *nonces* dentro do arquivo que está sendo cifrado. Consequentemente, não há necessidade de fazer ajustes em operações de *offset* no acesso aos blocos do arquivo, os quais ocorrem de forma alinhada ao tamanho das páginas de memória do SO e dos blocos do SAB, contribuindo para aumentar ainda mais os ganhos de desempenho. No modo CBC há o armazenamento do IVA no cabeçalho do arquivo, o que resulta na necessidade de ajustes de *offset* e o não alinhamento com as demais estruturas de armazenamento. Isto prejudica consideravelmente o desempenho do EncFS operando no modo CBC neste cenário que envolve a escrita aleatória de pequenas quantidades de blocos, o que pode ser percebido de forma mais clara na Figura 5.8 (a), com os ganhos alcançados pela utilização do modo CTR nas requisições de 4 KiB.

O cenário da leitura aleatória é onde praticamente não há ganhos ou perdas para requisições menores que 32 KiB. A justificativa está relacionada ao ganho modesto de desempenho oferecido na decifragem, já que o CBC também é paralelizável. Por oferecer um ganho menos significativo, neste cenário o que acaba dominando é o tempo de acesso a disco. Os ganhos só começam a aparecer quando o mecanismo de leitura antecipada começa a entrar em ação, o que pode ser percebido no caso de requisições maiores, a partir de 64 KiB. Em requisições de 512 KiB, o ganho se iguala ao da leitura sequencial, ficando na casa dos 15%.

Em versões anteriores foi testado o carregamento sob demanda dos *nonces* em memória, ou seja, eram carregados em memória somente os *nonces* relacionados aos blocos sendo acessados durante o tratamento de uma requisição. Apesar de ser uma abordagem que procurava reduzir o consumo de memória, ela mostrou-se problemática justamente no cenário de leitura aleatória, onde houve perda média de quase 9% no desempenho. Por este motivo, optou-se pelo carregamento prévio de todos os *nonces* do arquivo durante sua abertura, ajudando a evitar essas perdas mais significativas e alcançando ganhos em requisições maiores.

5.3.2 Vazão considerando operações simultâneas de leitura e escrita

A ferramenta `fio` também foi utilizada para medir os níveis de vazão alcançados em um cenário com acesso simultâneo realizado por múltiplos processos. Foram feitas análises simulando o acesso com um, dois, quatro e oito processos[7]. A ferramenta foi configurada para utilizar oito arquivos diferentes com tamanho de 640 MiB cada, sendo acessados de forma exclusiva por cada um dos processos.

Assim como os gráficos apresentados na subseção anterior, as Figuras 5.9, 5.10, 5.11 e 5.12 apresentam as porcentagens de ganho ou perda na vazão ao comparar o desempenho dos modos CTR e CBC, bem como os níveis de vazão alcançados. Os resultados obtidos nas medições com os diferentes números de processos foram agrupados no mesmo gráfico para facilitar a comparação. Foram feitos gráficos diferentes para cada uma das quatro operações medidas.

Na escrita sequencial, apresentada na Figura 5.9, pode-se notar o efeito cumulativo dos ganhos oferecidos pelo modo CTR. Conforme aumenta o número de processos, mais significativo

[7]Como o processador onde foram realizados os testes possui apenas 4 núcleos físicos, realizar o teste com mais processos não resultaria em níveis significativamente maiores de vazão.

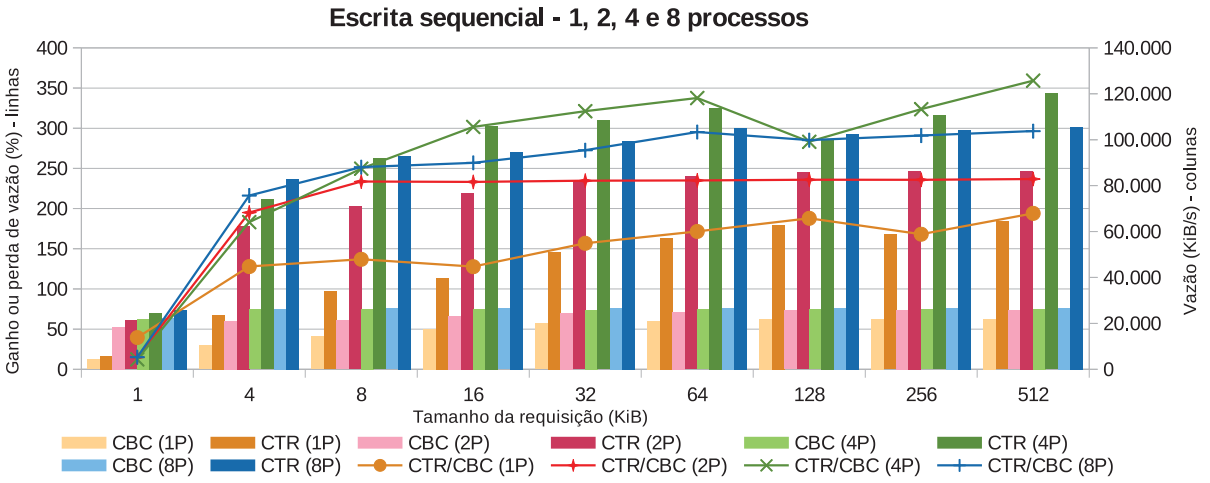


Figura 5.9: Níveis de vazão em escrita sequencial para 1, 2, 4 e 8 processos.

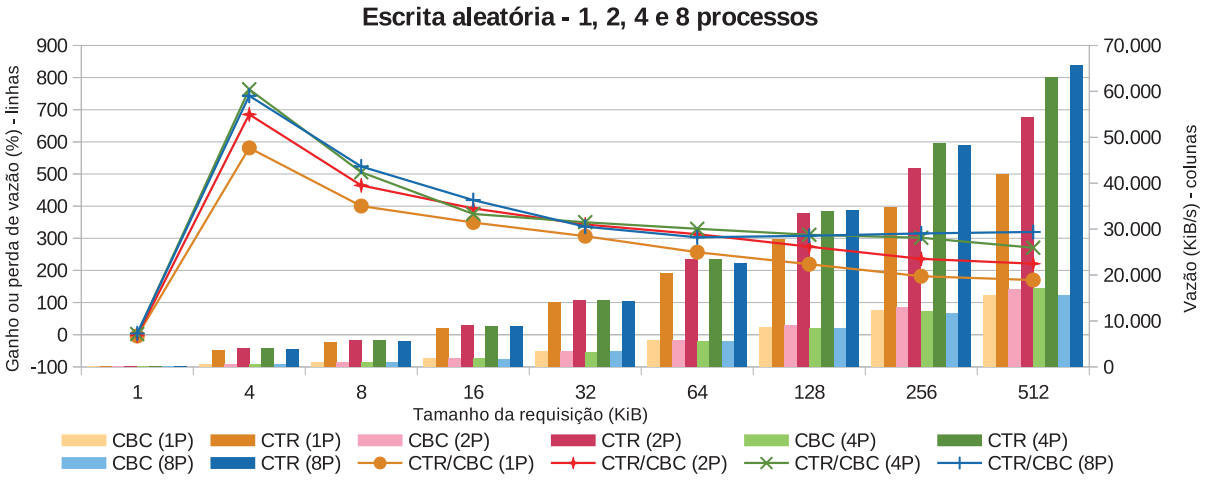


Figura 5.10: Níveis de vazão em escrita aleatória para 1, 2, 4 e 8 processos.

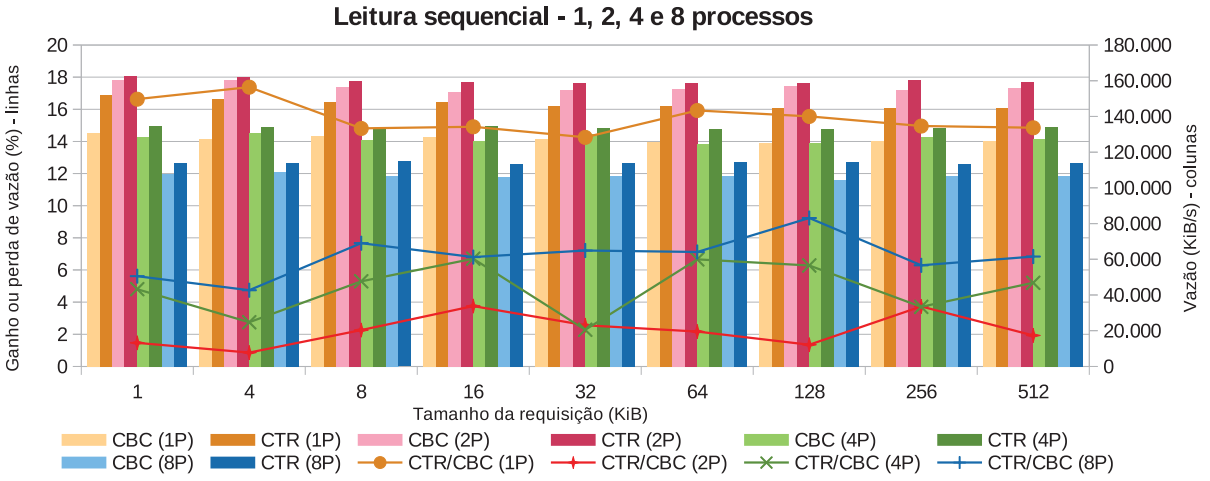


Figura 5.11: Níveis de vazão em leitura sequencial para 1, 2, 4 e 8 processos.

se torna o ganho. Na média, para um processo o ganho é de 159,79%; para dois 211,67% e para quatro 268,09%. Com oito processos começa a ocorrer uma concorrência maior pelo contexto

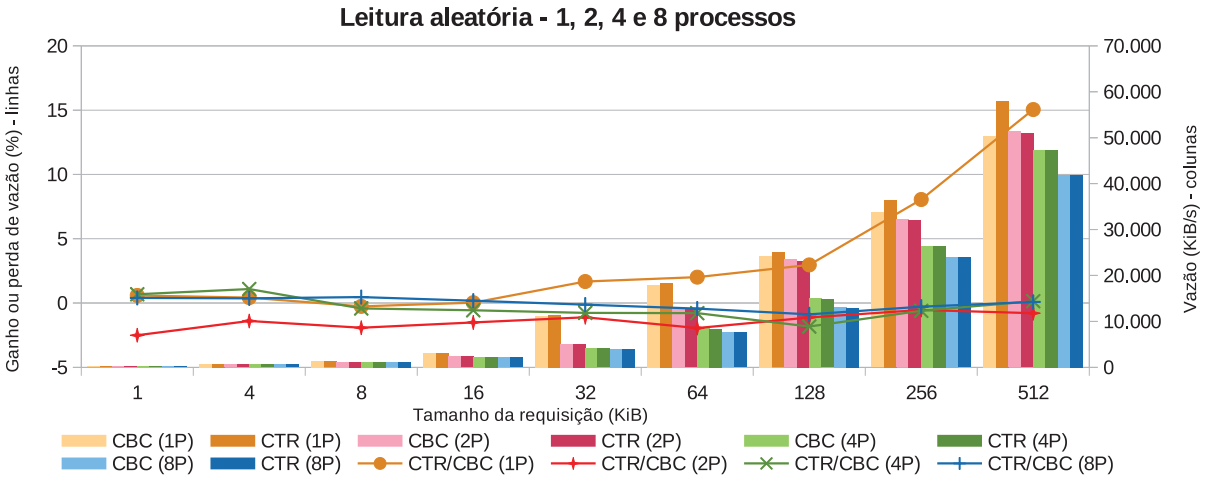


Figura 5.12: Níveis de vazão em leitura aleatória para 1, 2, 4 e 8 processos.

de cifragem e acesso a disco, refletindo-se numa redução nos ganhos que ficam na média em 246,33%.

A Figura 5.10 apresenta os resultados para a escrita aleatória. O comportamento com relação aos ganhos é semelhante ao da escrita sequencial, crescendo conforme aumenta o número de processos. Para um processo o ganho médio é de 216,92%; para dois 266,97% e para quatro 308,91%. Neste caso, o cenário com oito processos ainda apresenta aumento nos ganhos, ficando em 326,16%. Ao contrário do modo CTR, neste cenário o modo CBC já começa a apresentar diminuição nos níveis de vazão ao ser acessado por mais do que dois processos. Também é possível perceber o impacto no desempenho do SAC, ao operar no modo CBC, devido ao acesso não alinhado às demais estruturas de armazenamento. Ele fica mais claro nas operações que envolvem requisições de 4 KiB, o mesmo comportamento anteriormente observado na Figura 5.8 (a) e já explicado.

Já no caso da leitura sequencial, apresentada na Figura 5.11, o aumento na quantidade de processos acessando o SAC teve efeito contrário. Houve redução nos ganhos de desempenho. Para um processo, o ganho médio era de 15,48%, caindo para 2,22% com dois processos, 4,83% com quatro processos e 6,82% com oito processos. O aumento na quantidade de processos torna mais significativo o tempo de acesso a disco, reduzindo os ganhos provenientes da utilização do modo CTR. Comportamento semelhante ao da leitura aleatória descrito na subseção anterior.

Na leitura aleatória, apresentada na Figura 5.12, ao contrário do cenário com apenas um processo, onde os ganhos começavam a se manifestar a partir das requisições com 32 KiB, eles foram totalmente anulados ao se aumentar o número de processos. A situação de ter vários processos realizando acesso aleatório anula o efeito da leitura antecipada que começa a fazer efeito em requisições maiores e que pode ser observada no caso do acesso com somente um processo. Em compensação, mesmo com o aumento do número de processos, não houve aumento nas perdas, ficando praticamente zeradas. Para dois processos a perda média foi de 0,95%, com quatro processos 0,45% e com oito 0,19%.

As tabelas contendo os valores obtidos nesta análise com múltiplos processos podem ser vistas no Apêndice A.1.

5.3.3 Vazão com múltiplos contextos de cifragem

Todos os cenários descritos até agora foram baseados em análises de vazão onde os dados sendo acessados estavam armazenados em um disco rotacional. Em várias situações, como por exemplo na leitura sequencial a partir de dois processos, o gargalo claramente se torna o acesso a disco. Isto dificulta a análise de onde os níveis de vazão poderiam chegar e consequentemente da amplitude dos ganhos.

Para contornar a limitação imposta pelo tempo de acesso a disco, optou-se por utilizar um cenário com o armazenamento do SAB dentro do diretório `/run/shm`. Dessa forma, os arquivos do SAC acabam sendo armazenados em memória, eliminando a latência de acesso a disco. Também foi analisado um cenário com múltiplos processos visando mostrar os maiores níveis de vazão possíveis de serem alcançados. Neste cenário, o tamanho dos oito arquivos acessados pela ferramenta `fio` foi reduzido para 384 MiB por questões da limitação da quantidade de memória disponível e procurando evitar a ocorrência do *swap* de memória em disco.

A Figura 5.13 ilustra o primeiro caso testado de leitura sequencial. Nota-se que a partir de dois processos os níveis de vazão ficam estagnados, sendo que o esperado era que eles escalassem de acordo com o número de processos. O EncFS faz uso da biblioteca libfuse que por padrão é *multithread*. Consequentemente, o EncFS também se comporta como uma aplicação *multithread* onde cada *thread* se encarrega de tratar as requisições provenientes do módulo FUSE. Porém, devido ao fato do OpenSSL não ter suporte a *multithread* na utilização de contextos de cifragem simétrica e pelo EncFS ter sido codificado para utilizar somente um contexto de cifragem, ele precisa realizar *lock* sobre esse único contexto, procurando evitar situações de corrida. Assim, ocorre uma serialização na execução das *threads* sempre que uma função criptográfica precisa ser realizada, tornando-se um gargalo e impedindo que mais núcleos de processamento possam ser utilizados simultaneamente.

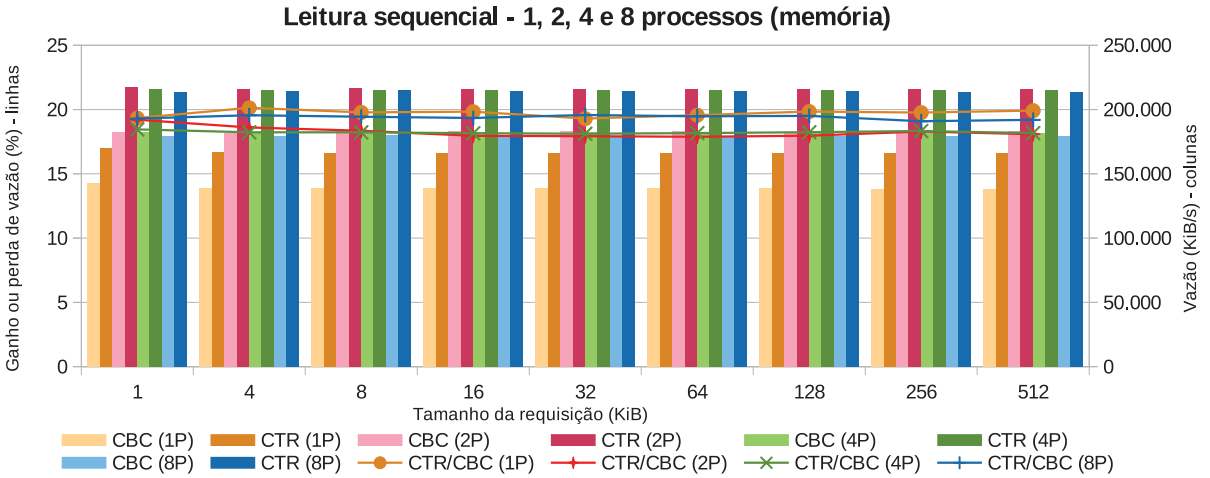


Figura 5.13: Exemplo de gargalo na vazão na leitura sequencial com arquivos armazenados em memória e acessados por múltiplos processos.

Como forma de contornar este problema, foi implementado um *pool* de contextos de cifragem. Desta forma, sempre que uma *thread* precisa executar uma função criptográfica, ela pega um contexto deste *pool* e o devolve após o término de sua utilização. Assim, várias *threads* podem executar funções criptográficas de forma simultânea, aproveitando melhor um ambiente multiprocessado. A implementação deste *pool* de contextos de cifragem foi realizada somente sobre a versão que utiliza o modo CTR.

As Figuras 5.14 e 5.15 ilustram os níveis de vazão alcançados nas operações de leitura e escrita sequenciais. Como o acesso neste cenário é em memória, optou-se por não exibir os resultados dos acessos aleatórios. Na operação de leitura sequencial, pode-se perceber claramente como o nível de vazão do modo CTR, utilizando o *pool* de contextos de cifragem, escala de acordo com a quantidade de processos. Enquanto no modo CBC a vazão fica estagnada por volta dos 182.000 KiB/s, no modo CTR ela chega a ficar na casa dos 763.000 KiB/s ao serem utilizados oito processos.

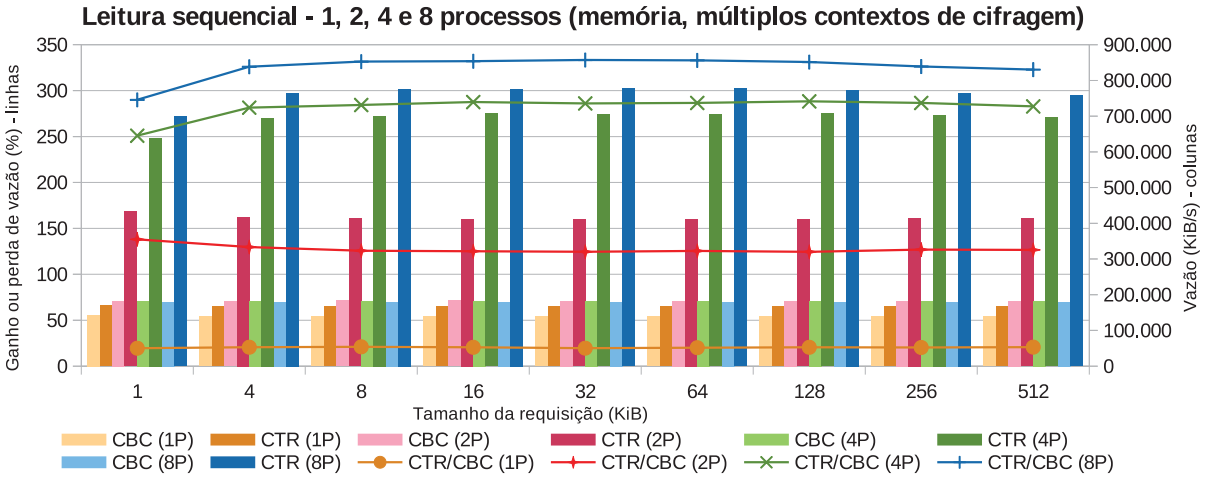


Figura 5.14: Níveis de vazão em leitura sequencial com arquivos armazenados em memória e utilizando múltiplos contextos de cifragem no modo CTR.

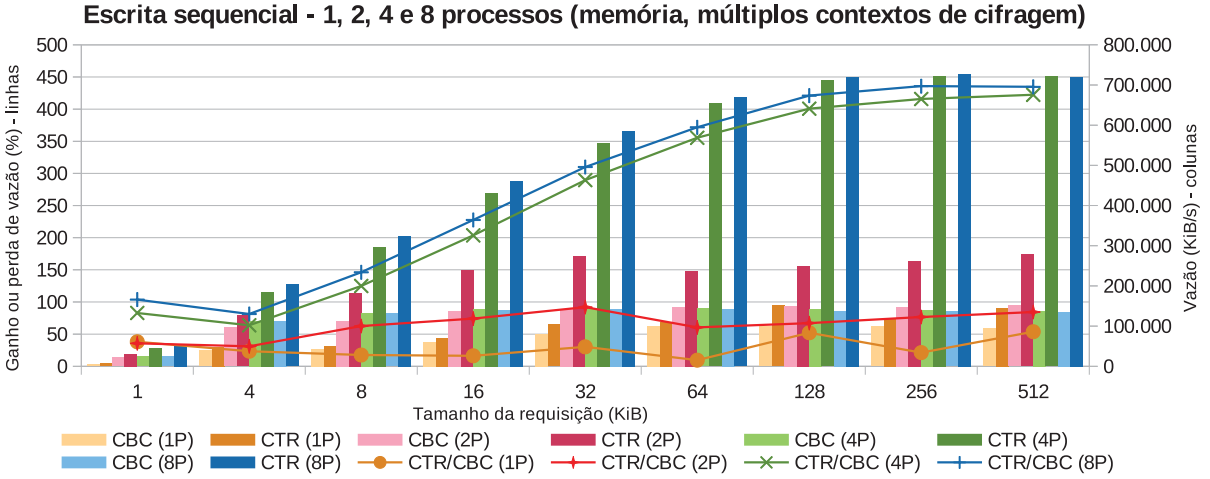


Figura 5.15: Níveis de vazão em escrita sequencial com arquivos armazenados em memória e utilizando múltiplos contextos de cifragem no modo CTR.

Comportamento parecido pode ser observado no caso da escrita sequencial que também escala a medida em que aumenta o número de processos. Os níveis de vazão atingem seu pico nas requisições maiores, por volta de 128 KiB, chegando a aproximadamente 720.000 KiB/s. É interessante notar que não há muita diferença nos níveis de vazão alcançados com quatro e oito processos. Como há somente quatro núcleos de processamento, isto acaba se tornado o gargalo. Também neste caso, por utilizar um único contexto de cifragem, o modo CBC fica com a vazão estagnada a partir de dois processos.

As tabelas contendo os valores obtidos nesta análise com múltiplos processos, acesso à memória e múltiplos contextos de cifragem podem ser vistas no Apêndice A.2.

5.3.4 Tempo de execução com *macrobenchmark*

A ferramenta `fio`, utilizada na seção anterior, permite avaliar os ganhos ou perdas de vazão ao se trabalhar sobre arquivos criados previamente antes das medições começarem. Por este motivo ela não contabiliza a sobrecarga de criação e expansão do arquivo. Essa questão se torna relevante se considerarmos a necessidade de gerenciamento dos arquivos exclusivos de *nonces*, os quais também precisam ser criados e expandidos à medida que os arquivos crescem. Portanto, pensou-se em um cenário de testes que procurasse contabilizar essa sobrecarga, realizando operações de criação, escrita e leitura de vários arquivos com tamanhos variados, distribuídos por vários diretórios.

Para a geração dos arquivos e criação das estruturas de diretórios foi escolhida a ferramenta `filebench`. A definição da quantidade de arquivos, tamanho médio, tipo de distribuição utilizada, entre outras podem ser definidas num arquivo de configuração específico. Consequentemente, esse arquivo pode ser reutilizado posteriormente para gerar os mesmos conjuntos de arquivos e diretórios, característica importante para se poder reproduzir os experimentos.

A ferramenta foi configurada e utilizada para gerar cinco cargas de trabalho diferentes visando medir o impacto no desempenho do SAC ao lidar com arquivos de variados tamanhos e quantidades. A quantidade de arquivos constante em cada carga, bem como o tamanho médio dos arquivos estão descritos a seguir:

- Carga 1: trabalha com uma quantidade significativa de arquivos, com tamanhos muito pequenos. Contém 10.000 arquivos com tamanho médio de 512 bytes;
- Carga 2: trabalha com uma quantidade significativa de arquivos, com tamanhos pequenos. Contém 8.000 arquivos com tamanho médio de 512 KiB;
- Carga 3: trabalha com uma quantidade menor de arquivos, com tamanhos medianos. Contém 200 arquivos com tamanho médio de 5 MiB;
- Carga 4: trabalha com uma quantidade menor de arquivos, com tamanhos razoáveis. Contém 50 arquivos com tamanho médio de 100 MiB;
- Carga 5: trabalha com uma quantidade pequena de arquivos, com tamanhos maiores. Contém 5 arquivos com tamanho médio de 1 GiB.

Apesar de a ferramenta dar suporte a uma linguagem de modelagem de cargas de trabalho e realizar medições sobre operações efetuadas sobre essas cargas, optou-se por utilizá-la apenas para gerar as cargas de trabalho. Para a escrita e leitura dos arquivos das cargas geradas pelo `filebench` foi utilizada a ferramenta `cp`; para a coleta dos tempos, a ferramenta `time`. Optou-se por essa abordagem pois o uso individual dessas ferramentas permite controlar melhor a sequência de comandos no cenário de medições. Principalmente permitir realizar o descarte de páginas de memória em *cache* entre as execuções de escrita e leitura dentro de uma mesma carga de trabalho sendo medida.

Sobre cada carga de trabalho foram executadas as seguintes operações:

- Operação 1: copiar arquivos da carga específica do diretório `/run/shm` para o SAC;
- Operação 2: copiar arquivos da carga específica do SAC para o diretório `/run/shm`;

A operação 1 é caracterizada como escrita sequencial e nesse cenário de medição contabiliza a sobrecarga envolvida na criação e expansão dos arquivos de *nonces* separados. A operação 2 caracteriza leitura sequencial, contabilizando a sobrecarga envolvida na leitura prévia dos *nonces* que é realizada durante a abertura do arquivo. Ambas operações também ajudam a medir a sobrecarga de gerenciamento dos *nnodes* quando uma quantidade significativa de arquivos é trabalhada. Os arquivos do SAC foram lidos e escritos no disco rotacional previamente descrito. O diretório `/run/shm` foi utilizado para neutralizar a interferência dos tempos de acesso de um segundo disco tanto na origem (caso da operação 1), quanto no destino (caso da operação 2). Cada operação foi aplicada a cada uma das cinco cargas de trabalho, sendo repetida 10 vezes e calculada a média aritmética simples.

A Figura 5.16 (a) e (b) apresenta os gráficos com os tempos obtidos nas operações 1 e 2 ao serem realizadas sobre os arquivos das cinco cargas anteriormente descritas.

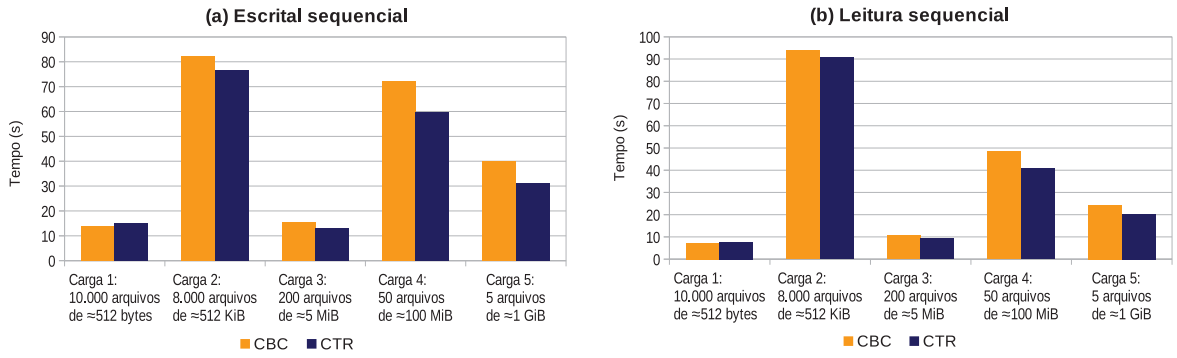


Figura 5.16: Comparativos de tempos para execução de operações de cópia sobre arquivos de diversos tamanhos e quantidades.

Os valores utilizados nos gráficos estão descritos nas Tabelas 5.5 e 5.6. Para ambos os modos de operação são apresentados os tempos mínimo, máximo, desvio padrão e média para cada uma das cinco cargas. A última coluna indica as porcentagens de aumento ou redução no tempo de execução. Como neste caso porcentagens negativas indicam diminuição no tempo de execução, valores menores do que zero foram coloridos em verde, indicando redução no tempo de execução e consequente melhora no desempenho. Valores maiores do que zero foram coloridos em vermelho, indicando aumento no tempo de execução e consequente piora no desempenho.

Tabela 5.5: Tempos e variação de tempos para operações de cópia de arquivos (escrita sequencial).

Carga			Tempo (s)								Variação Tempo CTR/CBC (%)
Número	Qtde. arquivos	Tam. dos arquivos	CBC				CTR				
			Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	10.000	512 bytes	13,87	14,2	0,1	14,08	14,84	15,41	0,17	15,17	7,72
2	8.000	512 KiB	80,84	83,25	0,6	82,35	75,64	77,74	0,68	76,66	-6,91
3	200	5 MiB	14,98	15,91	0,25	15,56	12,83	13,75	0,31	13,16	-15,43
4	50	100 MiB	67,82	76,99	2,68	72,22	58,3	61,74	1,36	59,77	-17,25
5	5	1 GiB	38,23	41,64	0,97	40,03	28,35	33,78	1,76	31,14	-22,2
Média			**43,15**	**46,4**	**0,92**	**44,85**	**37,99**	**40,48**	**0,86**	**39,18**	**-12,64**

Em ambos os casos de leitura e escrita sequencial, ao se trabalhar com uma grande quantidade de arquivos muito pequenos (média de 512 bytes), cenário caracterizado pela carga 1, é possível perceber o impacto causado pelo gerenciamento dos *nnodes*. O fato de se trabalhar com arquivos pequenos não permite que se obtenha ganhos significativos advindos das vantagens

Tabela 5.6: Tempos e variação de tempos para operações de cópia de arquivos (leitura sequencial).

Carga			Tempo (s)								Variação Tempo CTR/CBC (%)
Número	Qtde. arquivos	Tam. dos arquivos	CBC				CTR				
			Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	10.000	512 bytes	6,09	8,56	0,85	7,34	6,59	9,27	0,82	7,75	5,59
2	8.000	512 KiB	80,16	95,51	4,55	93,79	87,38	91,44	1,14	90,67	-3,33
3	200	5 MiB	10,42	10,66	0,08	10,51	9,09	9,28	0,05	9,19	-12,55
4	50	100 MiB	48,27	48,83	0,13	48,52	40,51	41,23	0,22	40,78	-15,95
5	5	1 GiB	23,77	25,1	0,53	24,3	19,74	21,15	0,48	20,23	-16,75
Média			**33,74**	**37,73**	**1,23**	**36,89**	**32,66**	**34,47**	**0,54**	**33,72**	**-8,59**

do modo CTR, os quais se manifestam somente quando quantidades maiores de dados são processados e os quais ajudam a amenizar o impacto causado. Na escrita sequencial o tempo de execução sofreu um aumento médio de 7,72% e na leitura 5,59%.

Contudo, ao se trabalhar com arquivos maiores, os ganhos advindos da utilização do modo CTR começam a se manifestar. Fato percebido ao se analisar os resultados das medições envolvendo os arquivos das cargas 2 até 5. Conforme o tamanho dos arquivos envolvidos nas medições vai aumentando, mais significativa vai se tornando a redução nos tempos de execução. Na escrita sequencial, a redução no tempo de execução vai de 6,91% (carga 2), chegando a 22,2% (carga 5). Na leitura sequencial, vai de 3,33% (carga 2) à 16,75% (carga 5).

É importante destacar que no cenário da escrita sequencial, além da sobrecarga do gerenciamento dos *nnodes*, também há a sobrecarga do gerenciamento dos arquivos exclusivos para armazenamento de *nonces* para arquivos maiores do que 64 KiB (16 blocos). Esta situação afeta os arquivos das cargas 2 até 5. Mesmo assim, consegue-se obter redução no tempo de execução utilizando-se o modo CTR.

Comparando-se os resultados da escrita sequencial mensurados pela ferramenta `fio` e os resultados medidos neste cenário, pode-se perceber o quanto o gerenciamento dos arquivos de *nonces* separados consegue reduzir os ganhos obtidos pela utilização do modo CTR. Porém, considerando-se um cenário de reescrita de arquivo, onde os arquivos de *nonces* já se encontram criados, os ganhos serão tão significativos quantos os demonstrados pela ferramenta `fio`. O mesmo se aplica aos ganhos alcançados no cenário de escrita aleatória, pois a mesma normalmente é realizada sobre arquivos preexistentes e cujos arquivos de *nonces* também já estariam pré-criados.

5.3.5 Latência das operações de leitura e escrita

Para medir os níveis de latência foi utilizada a ferramenta `strace`, a qual permite contabilizar os tempos de diversas chamadas de sistema ao SO. Esta medição foi aplicada sobre as operações de cópia de arquivos das cinco cargas de trabalho descritas na subseção anterior. Foram contabilizadas todas as chamadas de sistema envolvidas nas operações de cópia. As medições foram repetidas 10 vezes e calculada a média aritmética simples.

Na Figura 5.17 (a) e (b) estão graficados os níveis de latência referentes às operações de cópias dos arquivos para cada uma das cinco cargas de trabalho, caracterizando operações de escrita e leitura sequenciais. Na sequência, nas Tabelas 5.7 e 5.8, são apresentados os valores obtidos. Como a medição da latência também é baseada em tempo, os ganhos ou perdas de tempo, bem como a colorização dos valores da última coluna das tabelas, seguem a mesma lógica utilizada na subseção anterior.

No caso da escrita utilizando os arquivos da carga 1, é difícil perceber no gráfico o aumento de latência do modo CTR comparado ao CBC. Na Tabela 5.7, pode-se ver que neste

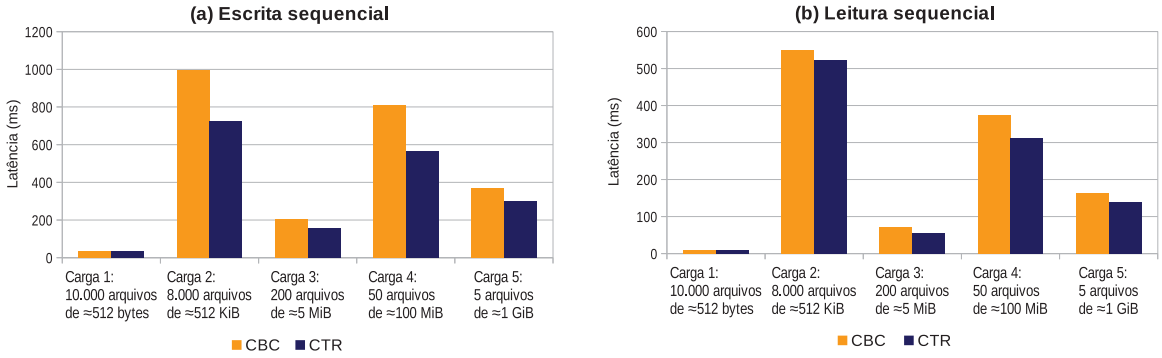


Figura 5.17: Níveis de latência envolvidos em operações de cópias de arquivos.

Tabela 5.7: Níveis de latência envolvidos em cópia de arquivos (escrita sequencial).

Carga			Tempo (ms)								Variação Latência CTR/CBC (%)
Número	Qtde. arquivos	Tam. dos arquivos	CBC				CTR				
			Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	10.000	512 bytes	29,19	37,73	2,59	33,27	31,34	40,52	2,89	35,97	8,12
2	8.000	512 KiB	963,03	1.021,68	17,33	997,02	694,51	753,33	18,44	726,85	-27,1
3	200	5 MiB	182,02	222,9	12,47	205,42	137,85	178,18	12,21	157,91	-23,13
4	50	100 MiB	768,08	874,56	34,93	811,13	490,56	631,83	42,99	563,93	-30,48
5	5	1 GiB	320,03	460,3	40,15	366,94	238,84	344,63	33,47	300,76	-18,04
Média			**452,47**	**523,43**	**21,5**	**482,76**	**318,62**	**389,7**	**22**	**357,08**	-26,03

Tabela 5.8: Níveis de latência envolvidos em cópia de arquivos (leitura sequencial).

Carga			Tempo (ms)								Variação Latência CTR/CBC (%)
Número	Qtde. arquivos	Tam. dos arquivos	CBC				CTR				
			Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	10.000	512 bytes	7,78	12,77	1,33	10,7	9,74	11,82	0,62	10,64	-0,63
2	8.000	512 KiB	459,02	611,99	53,08	548,98	479,48	572,78	28,38	522,56	-4,81
3	200	5 MiB	59,75	78,81	5,89	70,79	41,56	69,81	8,82	56,4	-20,33
4	50	100 MiB	347,06	411,18	20,45	373,63	287,96	334,22	16,59	311,45	-16,64
5	5	1 GiB	151,94	176,95	9,05	163,53	119,38	151,09	9	137,65	-15,83
Média			**205,11**	**258,34**	**17,96**	**233,52**	**187,62**	**227,95**	**12,68**	**207,74**	-11,04

caso em específico houve um aumento de 8,12%. Este resultado corrobora o aumento no tempo de execução observado na subseção anterior quando se mediu os tempos de escrita sobre arquivos desta carga. A questão de se trabalhar com pequenas quantidades de dados, contidos em arquivos desta carga, não chega a oferecer ganhos suficientes advindos da utilização do modo CTR a ponto de compensar a sobrecarga com a alocação de *nnodes* para cada um dos 10.000 arquivos criados. A alocação de *nnodes* ocorre sobre estruturas de dados que estão mapeadas em disco, portanto há a sobrecarga da escrita em disco, além do incremento do contador geral do SAC, que também é mantido mapeado.

Nas medições de latência envolvendo as demais cargas (2 a 5), há redução nos valores ao se utilizar a implementação que utiliza o modo CTR. No caso da escrita, essa redução oscila entre 27,1% e 30,48% nas cargas de 2 a 4, reduzindo para 18,04% no caso da carga 5. Considerando todas as cargas, a média de redução na latência fica em 26,03%.

No caso da leitura, a latência para os arquivos da carga 1 praticamente não se altera entre os dois modos, pois apresenta ganho de apenas 0,63%. Como na leitura não há a necessidade de alocação de *nnodes* e os mesmos já se encontram carregados em memória, o impacto é amenizado. A redução na latência começa a ser mais significativa somente com arquivos das

cargas 3 a 5, pois alcança os valores de 20,33%, 16,64% e 15,83%, respectivamente. Na média geral, considerando os valores de todas as cargas, a redução na latência é de 11,04%.

5.4 Considerações finais

As análises de desempenho realizadas mostraram que a implementação do modo CTR trouxe ganhos significativos de desempenho nas operações de reescrita sequencial e aleatória de arquivos. Em média, esse ganho chegou a atingir 268,09% na reescrita sequencial e 326,16% na escrita aleatória num cenário com acesso de até oito processos e arquivos armazenados em disco rotacional.

Na leitura sequencial os ganhos foram mais modestos, em média por volta de 15,48%, devido ao fato da cifragem no modo CBC também ser paralelizável e consequentemente não haver grandes diferenças de desempenho entre os dois modos. Esta característica também se reflete na leitura aleatória, onde praticamente não houve perdas nem ganhos de desempenho. Somente com requisições de tamanhos maiores, acima de 128 KiB, os ganhos se aproximam da leitura sequencial.

Nos cenários em que se trabalha escrevendo e lendo arquivos pequenos, a sobrecarga de gerenciamento dos *nonces* é mais significativa e pode levar a pequenas perdas de desempenho. Para arquivos com tamanho médio de 512 bytes, na escrita sequencial essa perda é de aproximadamente 7% e, na leitura sequencial, 5%. Ao se trabalhar com arquivos maiores, a sobrecarga de gerenciamento dos *nonces* torna-se menos significativa e os ganhos da utilização do modo CTR começam a se tornar mais significativos.

As análises também ajudaram a comprovar a limitação imposta pela utilização de um único contexto de cifragem implementado por padrão no EncFS. Na prática, esse contexto único limita as funções criptográficas no sentido de apenas utilizar um núcleo de processamento por vez. A implementação de um *pool* de contextos de cifragem ajudou a contornar esta limitação e a demonstrar o potencial de ganho, permitindo utilizar de forma simultânea mais núcleos de processamento nas funções criptográficas.

É interessante ressaltar que os resultados obtidos nestas análises são importantes no sentido de comprovar que a implementação das técnicas relacionadas à geração, armazenamento e manipulação de *nonces* não causaram impacto negativo e significativo no desempenho do SAC. Pelo contrário, em alguns cenários houve melhoras consideráveis de desempenho.

Outra questão a ser destacada é que não foram exploradas vantagens específicas do modo CTR, como a geração antecipada de máscaras de cifragem e o processamento das funções criptográficas em GPU. Com a questão dos *nonces* resolvida pela implementação realizada neste trabalho, há grande potencial para que essas características específicas do modo CTR sejam exploradas e tragam ganhos ainda mais significativos, principalmente nas operações de leitura onde eles foram modestos. A exploração dessas vantagens é justamente o assunto abordado no capítulo seguinte.

Capítulo 6

Sistema de arquivos criptográfico com processamento em GPU

Realizar o processamento das funções criptográficas de um SAC em GPU pode ser promissor no sentido de se alcançar níveis maiores de vazão de dados e uma utilização mais eficiente de CPU nos processos de leitura e escrita de arquivos. Uma das ênfases deste trabalho está em preparar um SAC para cifragem eficiente em GPU com a utilização do modo CTR. Neste sentido, a primeira etapa deste processo, relacionada à implementação do modo CTR no contexto de SACs, foi abordada no capítulo anterior. Este capítulo corresponde a segunda etapa, a qual envolve o processamento em GPU e onde são apresentadas técnicas para o gerenciamento dos contextos de cifragem utilizados pela biblioteca WAESlib.

As técnicas apresentadas demonstram como utilizar os contextos de cifragem procurando fazer com que as máscaras de cifragem sejam produzidas com o nível de antecedência adequado e estejam prontamente disponíveis nos momentos em que forem necessárias para efetivamente cifrar e decifrar blocos de dados. Com esse objetivo, a definição dos contextos deve tentar explorar a forma como os arquivos armazenados no SAC são acessados, o que pode ser feito, por exemplo, analisando-se questões de localidade e padrões de acesso. Essas técnicas representam uma importante contribuição deste trabalho.

Para atingir esse objetivo utilizamos uma versão mais recente do *kernel* de GPU WAES [14] e sua biblioteca WAESlib, pois ela permite utilizar recursos de prioridades na produção especulativa de máscaras para cifragem CTR, mostrando-se mais adaptada ao funcionamento de um SAC.

A Seção 6.1 apresenta trabalhos correlatos que analisam a aceleração do cifrador AES em GPU e na sequência são descritos alguns trabalhos que exploram a aceleração das funções criptográficas em GPU aplicadas especificamente a SACs. A Seção 6.2 apresenta o WAES e a WAESlib. Na Seçao 6.3 são discutidas técnicas que visam utilizar de forma eficiente os recursos oferecidos pela WAESlib. A Seção 6.4 apresenta alguns detalhes da implementação envolvendo a integração do EncFS, operando no modo CTR, e a WAESlib, bem como a aplicação das técnicas discutidas na Seção 6.3. A Seção 6.5 apresenta uma análise inicial de desempenho da implementação resultante. Considerações finais acerca dos assuntos abordados neste capítulo estão presentes na Seção 6.6.

6.1 Aceleração do AES em GPU e utilização do processamento em GPU aplicado a SACs

A utilização do processamento em GPUs para acelerar a cifragem do AES já foi extensamente estudada. Ao longo das pesquisas, foram estudadas e criadas diferentes técnicas e métodos visando melhor utilização dos recursos das GPUs, bem como otimizar a forma como o algoritmo pode ser implementado.

Em [4] foi analisada qual a granularidade dos dados a serem cifrados resulta em melhor desempenho. Foi analisado o processamento de 4 e 16 bytes por *thread*. Também foi analisado o armazenamento das T-box na *constant* e *shared memory*. Em [5] foi analisado o uso de *streams* para se conseguir a execução simultânea de processamento e cópia dos dados entre a CPU e GPU. Foram analisadas questões como a geração das subchaves do AES em tempo de processamento, bem como seu armazenamento na *texture* e *shared memory*.

Em [6] foi proposto um método onde uma *thread* era mapeada para processar mais do que um bloco de dados de forma iterativa. Em [7] foi proposto um método para reorganizar os dados a serem cifrados na memória da CPU antes de serem copiados para a memória da GPU. Essa reorganização visava permitir que o acesso feito pelas *threads* à global *memory* pudesse ser feito de forma agrupada (*coalesced*). Em [8] foram sugeridos o uso de instruções nativas para a realização de multiplicações e o uso da diretiva `unroll loops` para reduzir a quantidade de registradores utilizados.

Algumas publicações mais recentes analisaram arquiteturas atuais de GPUs e recursos novos. Em [9] foi proposto o armazenamento das subchaves nos registradores e a utilização da operação `warp shuffle` para permitir o compartilhamento do conteúdo desses registradores entre *threads* pertencentes ao mesmo *warp*. Esse estudo chama atenção pois entre as soluções comparadas está o WAES. No comparativo de desempenho, o WAES ficou em segundo lugar, sendo apenas 4% mais lento do que a solução proposta no artigo. Há que se considerar que a solução proposta no artigo utiliza recursos mais atuais de GPUs NVIDIA, os quais não estavam disponíveis na época da criação do WAES.

Com relação a utilização dos recursos de processamento em GPU aplicado a sistemas de arquivos, existem publicações com duas abordagens diferentes de integração: para aplicações que rodam no espaço do usuário e para aplicações que rodam no espaço do *kernel*. As referências [8], [87], [88], [10] e [11] se referem a soluções no espaço do usuário. As referências [89], [90], [12], [91] e [13] apresentam soluções para o espaço do *kernel*.

No projeto Engine-CUDA [87] foi implementada uma *engine* OpenSSL capaz de executar operações de cifragem simétricas. Essas operações foram implementadas utilizando CUDA. Através dessa *engine* foi possível criar contextos de cifragem normais OpenSSL, porém que utilizavam a GPU para executar os algoritmos de cifragem. O projeto Engine-CUDA serviu de base para o desenvolvimento do trabalho feito em [8]. Nele, a *engine* foi aprimorada e estendida. Uma série de otimizações foram feitas na implementação do AES. Foram implementados também os cifradores DES, Blowfish, IDEA, Camellia e CAST5. Resultados mostraram que o processamento em GPU se torna efetivo acima dos 16 KiB e chega ser oito vezes melhor ao se aproximar dos 8 MiB.

CrystalGPU [88] é um *framework* desenvolvido com o objetivo de facilitar a integração em aplicações do processamento em GPU. A aplicação prática do *framework* foi estudada através de sua integração a outra biblioteca denominada StoreGPU. A mesma foi utilizada para acelerar funções de *hash* utilizadas em sistemas de armazenamento de dados que utilizam a técnica de endereçamento baseado em conteúdo (*addressable content storage*). Foram feitas análises com

blocos de dados variando de 64 KiB até 32 MiB. Em blocos menores o ganho ficou próximo de 1,6 vezes e chegou a ser seis vezes mais rápido em blocos maiores.

Já em [10], o CrystalGPU também serviu como base de integração para o desenvolvimento de um sistema de arquivo baseado no FUSE com recursos de confiabilidade, denominado CRSFS. O processamento em GPU envolvia a execução das operações de codificação do algoritmo *Cauchy Reed-Solomon* (CRS). Mais tarde, em [11], o mesmo sistema de arquivos foi aperfeiçoado, adicionando o recurso de cifragem dos dados também com processamento em GPU. O algoritmo utilizado foi o AES operando nos modos ECB e CBC. Os resultados indicaram vantagem de processamento em CPU para dados com tamanhos até 4 KiB e em GPU para dados acima dos 16 KiB. Também foram avaliados cenários com a utilização de mais GPUs simultaneamente.

O OpenBSD *Cryptographic Framework* (OCF) [89] é um *framework* originalmente desenvolvido com o objetivo de permitir a utilização dos recursos de processamento de placas aceleradoras. Mais especificamente, placas com recursos de processamento para funções criptográficas. O diferencial desse *framework* é ser diretamente acessível a aplicações que rodam no espaço do *kernel*. O trabalho feito em [90] fez uso da versão para Linux do OCF visando integrar os recursos de processamento em GPU a esse *framework*. Através dessa integração foi possível executar em GPUs NVIDIA operações de cifragem simétrica utilizando AES no modo ECB.

Gdev [12] é um sistema de *runtime* criado para ser executado no espaço do *kernel* visando gerenciar a utilização da GPU de forma análoga a como são gerenciados demais processos que são executados na CPU. Além de rodar no espaço do *kernel*, seu principal diferencial é poder controlar a GPU utilizando um *driver* de código aberto (Nouveau), sem depender dos *drivers* proprietários e das bibliotecas de acesso que rodam no espaço do usuário. Umas das aplicações práticas demonstradas foi a adaptação do eCryptfs para realizar cifragem em GPU através do Gdev. Para operações de gravação o ganho chegou a ser de até duas vezes.

O GPUStore [91][13] é um sistema de *runtime* e *framework* criado com o objetivo de facilitar a integração e tornar eficiente a utilização de processamento em GPU para sistemas de armazenamento de dados que rodam no espaço do *kernel*. Em [13], como aplicação prática, o GPUStore foi utilizado para acelerar três aplicações que rodam no *kernel* do Linux: o dm-crypt, o eCryptfs e a solução de RAID baseado em software oferecida pelo md. Para cifragem de dados acima dos 256 KiB, o desempenho chegou a ser 36 vezes melhor.

6.2 WAES e WAESlib

O WAES (*Warped AES*) [14] é a implementação de um algoritmo de alto desempenho para processamento paralelo heterogêneo utilizado para cifragem de dados usando GPU. No WAES foi implementado o AES com chaves de 128 e 256 bits no modo de operação CTR. A parte heterogênea do WAES refere-se a possibilidade de realizar todas as etapas do modo de operação CTR na GPU ou somente a parte final de `XOR` com o texto a ser cifrado ou decifrado na CPU.

O WAES explora a característica do modo de operação CTR para implementar uma técnica denominada cifragem especulativa (*speculative encryption*). A partir do momento em que se tem a chave de cifragem e o *nonce*, o WAES pode computar e preencher antecipadamente *buffers* com dados cifrados. Dessa forma, esses dados ficam disponíveis antes mesmo de serem requisitados pela aplicação, sendo denominados máscaras de cifragem.

Com relação ao gerenciamento de memória da CPU e GPU, o WAES utiliza um recurso do ambiente CUDA para mapear a memória da CPU no espaço de endereçamento da GPU e vice-versa. Essa técnica também requer o uso de memória não-paginada. Com isso, um *kernel*

WAES pode gerar dados nesses endereços mapeados, os quais ficam diretamente acessíveis na CPU.

O WAES possui um modo de operação denominado *double buffering*, onde a operação de `XOR` na CPU pode ser realizada simultaneamente com a geração de mais máscaras de cifragem na GPU. Também é possível realizar a agregação de *buffers*. Pode-se agregar mais de um *buffer* do mesmo contexto de cifragem ou de contextos diferentes, os quais serão processados numa mesma ativação do *kernel* WAES. A agregação de *buffers* pequenos e seu processamento numa única ativação do *kernel* WAES permite que eles fiquem disponíveis antecipadamente para serem posteriormente processados na CPU, conforme forem requisitados.

A técnica de agregação de *buffers* diminui a latência ao mesmo tempo em que permite aumentar a utilização dos núcleos de processamento da GPU, consequentemente obtendo-se uma melhor vazão de dados. Particularmente, esse recurso de agregação parece promissor no caso de SACs que processam pequenas quantidades de dados por requisição, geralmente ficando na casa dos 4 KiB. Dessa forma, é possível compensar a latência adicionada pela necessidade de transferência de dados entre CPU e GPU.

Para se utilizar os recursos de cifragem do WAES anteriormente descritos em uma aplicação, pode-se fazer uso da biblioteca denominada WAESlib. A chamada a essa biblioteca se encarrega de preparar o ambiente CUDA e disparar o *kernel* WAES para que o mesmo comece a pré-computar as máscaras de cifragem. As chamadas para a geração de máscaras feitas através da WAESlib são assíncronas. Dessa forma, uma *thread* que faz uma chamada dessas pode continuar executando outras operações sem ficar bloqueada. Ela pode até mesmo fazer outras chamadas para geração de novas máscaras.

A WAESlib também oferece um recurso de prioridades que pode ser utilizado para controlar a ordem de produção das máscaras de cifragem. Contextos definidos com maior prioridade tem suas máscaras produzidas antes, sendo que a biblioteca se encarrega de automaticamente readequar para uma prioridade maior os contextos que foram deferidos por terem inicialmente uma prioridade menor. No contexto de SACs, esse recurso pode ser bastante útil no sentido de tentar ordenar a produção das máscaras de acordo com a ordem em que os blocos de um arquivo são acessados.

6.3 Técnicas para utilização eficiente da WAESlib

Para se compreender algumas das técnicas propostas neste trabalho, as quais visam utilizar de forma eficiente a WAESlib no contexto de um SAC, é necessário descrever brevemente seu funcionamento[1]. A utilização geral da biblioteca WAESlib está baseada em alguns passos básicos: inicialização do ambiente, configuração de chaves de cifragem, definição ou redefinição de contextos de cifragem, acionamento dos contextos para cifragem ou decifragem de dados e finalização do ambiente.

Na inicialização, são repassadas algumas informações à biblioteca, entre elas a quantidade de contextos a serem utilizados e tamanho dos segmentos. A quantidade de contextos define quantos contextos serão criados pela WAESlib e estarão disponíveis para utilização na aplicação. O tamanho dos segmentos define o tamanho das máscaras de cifragem geradas. No contexto de SACs, este tamanho é igual ao tamanho dos blocos de cifragem dos arquivos.

[1]Uma breve descrição das funções disponibilizadas pela API da biblioteca WAESlib pode ser consultada no Apêndice B.

A função de definição de chave permite criar chaves que poderão ser utilizadas posteriormente na função de definição de contextos. Nela são repassadas informações como número de identificação da chave e seu tamanho.

O funcionamento geral da WAESlib está centrado nas funções de definição de contextos de cifragem e nas funções que fazem o seu acionamento para efetivamente cifrar ou decifrar dados.

Na utilização da função de definição de contextos são informados o número de identificação do contexto sendo definido, o número de identificação da chave previamente definida, o *nonce* e um número de prioridade. A chamada a essa função é assíncrona, liberando a aplicação para executar outras tarefas. Após a chamada da função, a biblioteca pode acionar o *kernel* WAES para que o mesmo comece a computar antecipadamente na GPU as máscaras de cifragem.

Internamente a prioridade é utilizada pela WAESlib para agregar e definir a ordem na produção das máscaras. À medida em que as máscaras vão sendo geradas, elas são transferidas para a memória de sistema, ficando disponíveis para serem utilizadas nas funções de cifragem e decifragem de dados.

As outras duas principais funções são utilizadas para instruir a WAESlib a cifrar e decifrar dados. Na sua chamada são passadas informações como o número de identificação do contexto, o *buffer* contendo os dados a serem cifrados ou decifrados e o seu tamanho. São funções síncronas que fazem com que a WAESlib utilize a máscara de cifragem previamente calculada para o contexto em questão, disponível na memória de sistema, para efetivamente cifrar ou decifrar os dados. Essa etapa final consiste numa operação de `XOR` entre os dados e a máscara gerada, sendo realiza na CPU.

Basicamente, a utilização da biblioteca resume-se num processo que envolve chamar a função de definição de contextos com a finalidade de dar início a produção das máscaras de cifragem e posteriormente chamar as funções que aplicam essas máscaras geradas. Apesar da sua utilização simplificada, há alguns desafios envolvidos neste processo. Eles dizem respeito principalmente a como dividir, definir e utilizar de forma eficiente esses contextos de cifragem nas operações de escrita e leitura tanto sequencial quanto aleatória.

Com relação a divisão dos contextos, a técnica proposta neste trabalho está baseada na criação de *pools* de contextos. Define-se uma unidade de alocação específica de contextos. A quantidade total de contextos criados na inicialização da WAESlib é dividia por esta unidade, resultando na quantidade total de *pools* disponíveis. Cada *pool* recebe uma numeração lógica que começa em zero e é incrementada através da fórmula $1 << log_2(y)$, onde y corresponde ao tamanho da unidade de alocação. De acordo com a quantidade necessária de contextos para uma determinada finalidade dentro da aplicação, pode-se agregar um ou mais *pools*.

Nas chamadas das funções da WAESlib que tem como parâmetro o número de identificação do contexto, a aplicação utilizará como identificador do contexto a concatenação do número lógico do *pool* com um número sequencial correspondente aos contextos contidos dentro do *pool*, iniciando em zero e indo até a quantidade de contextos contidos no *pool* menos um. Dessa forma, os bits menos significativos identificam o contexto dentro do *pool*.

Com relação a utilização dos *pools* de contextos, identificou-se a necessidade de se utilizar pelos menos dois tipos diferentes: um *pool* único de contextos de cifragem, em nível do SAC, utilizado nas operações de escrita sequencial e aleatória; e vários *pools* de contextos para decifragem, um por arquivo aberto, utilizados nas operações de leitura sequencial e aleatória. A ideia de funcionamento desses *pools* está descrita nas duas subseções seguintes.

6.3.1 *Pool* de contextos para escrita

Visando esconder a latência envolvida no processo de geração das máscaras na GPU, bem como sua transferência da memória do dispositivo para a memória de sistema, é essencial que o disparo da geração das máscaras ocorra o mais cedo possível. Procurando atingir esse objetivo nos casos da escrita sequencial e aleatória, este trabalho propõe a utilização de um único *pool* de contextos utilizado para a geração de máscaras que podem ser utilizadas nos processos de cifragem dos blocos. Consequentemente, elas pode ser utilizadas nos processos de escrita e reescrita de blocos de todos os arquivos do SAC.

Utilizando como exemplo um *pool* contendo oito contextos, quando o SAC é montado, oito contextos serão alocados para esse *pool*. Copia-se o valor atual do contador do SAC e definem-se os contextos contidos no *pool* utilizando-se um *nonce* correspondente ao valor desse contador, o qual é incrementado a cada novo contexto definido. Respeitando a necessidade de se manter reservados os bits menos significativos do contador e considerando um SAC que utilize blocos de 4 KiB, o contador sofre incrementos no valor de 256.

O indicador do início da fila é utilizado para guardar a identificação do contexto contendo a máscara mais antiga gerada. Ao final desse processo, ilustrado na Figura 6.1 (a), o SAC terá a disposição uma quantidade correspondente ao tamanho do *pool* em máscaras prontas para serem utilizadas nos processos de cifragem de blocos.

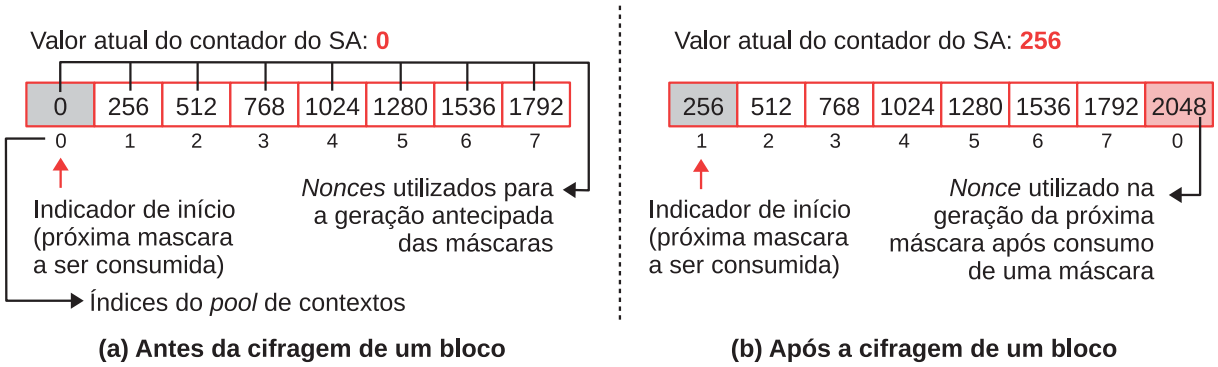


Figura 6.1: Funcionamento do *pool* de contextos para escrita.

Porém, para não ocorrer o desperdício de *nonces* e evitar a necessidade de armazenar temporariamente os *nonces* utilizados na geração das máscaras, o contador do SAC só é efetivamente incrementado quando uma máscara é utilizada no processo de cifragem. O *nonce* a ser salvo para ser utilizado no processo futuro de decifragem do bloco é obtido a partir do valor do contador do SAC quando ocorre a utilização da máscara. Logo após, o contador é incrementado, passando a corresponder ao valor utilizado na geração da máscara do contexto subsequente. O indicador do início da fila é então deslocado.

Após o consumo da máscara, o contexto que continha a máscara recentemente utilizada é reutilizado para disparar a geração de uma nova máscara. Assim, todos os contextos são mantidos sempre com uma máscara nova. A prioridade utilizada nesta redefinição de contexto pode ser a menor possível, pois a máscara a ser produzida só será consumida após todas as outras. O valor do *nonce* a ser utilizado na geração da próxima máscara pode ser obtido através da fórmula $x = y + (1 << w) * z$, onde x corresponde ao próximo *nonce*, y o valor atual do contador do SAC, w a quantidade de bits reservados do modo CTR e z o tamanho do *pool* de contextos de escrita. A Figura 6.1 (b) ilustra o estado do *pool* após o consumo de uma máscara, seguido da geração de uma máscara nova.

6.3.2 *Pool* de contextos para leitura

O *pool* de contextos utilizado para decifragem, e consequentemente nos processos de leitura, funciona como uma janela de máscaras que é deslocada de acordo com a região do arquivo sendo lida. Nesta técnica, cada arquivo aberto possui seu *pool* de contextos exclusivo cujo disparo para geração de máscaras ocorre de acordo com o número do bloco sendo acessado. A janela é posicionada sobre o primeiro bloco lido, sendo disparada a geração de máscaras para os blocos subsequentes de acordo com o tamanho do *pool*. Conforme os blocos vão sendo lidos e as máscaras consumidas nos processos de decifragem, a janela vai sendo deslocada e novas máscaras são geradas.

A Figura 6.2 ilustra o comportamento do *pool* de contextos contendo as máscaras geradas num processo de leitura sequencial, começando no início do arquivo. Também tomando como exemplo um *pool* contendo oito contextos, quando o arquivo é aberto, os *nonces* dos oito primeiros blocos são utilizados para definir os contextos de decifragem que começam a produzir as máscaras[2].

Figura 6.2: *Pool* de contextos para leitura e deslizamento da janela em leitura sequencial.

O recurso de prioridades oferecido pela WAESlib pode ser explorado para ditar a ordem de produção dessas máscaras. Assim, ao definir cada um dos contextos da janela, pode-se definir o contexto contido no início da janela com a maior prioridade possível, diminuindo gradativamente essa prioridade na definição dos contextos subsequentes. As máscaras necessárias para decifrar os primeiros blocos vão ficando prontas antes das máscaras dos blocos subsequentes.

Na ilustração da Figura 6.2, após o consumo da máscara referente ao bloco zero, pode-se disparar a produção da máscara para o bloco oito com a menor prioridade possível, já que a tendência é ela ser utilizada somente após o consumo das máscaras dos blocos anteriores. Numa leitura puramente sequencial, esse processo vai se repetindo, com o consumo da máscara do início da janela e a subsequente produção de uma nova máscara ao seu final.

A técnica da janela deslizante também pode ser aplicada para acessos aleatórios. Neste caso, seu início é deslocado sempre para a posição correspondente ao primeiro bloco sendo lido. Quando acontece este deslocamento, a princípio podem ocorrer duas situações: (i) a janela é totalmente deslocada para uma posição anterior a que estava antes, ou seja, $(x - y) > z$, onde x é posição inicial anterior da janela, y a nova posição inicial e z o tamanho da janela; (ii) a janela

[2]Os quadros em preto representam os blocos do arquivo e o seu conteúdo o número do bloco. Os quadros em vermelho representam a janela de máscaras e os números representam os *nonces* utilizados para gerar as máscaras.

é deslocada totalmente para frente, onde $(y - x) > z$. Ambas as situações estão ilustradas na Figura 6.3 (a) e (b), respectivamente. Como as novas posições iniciais estão totalmente fora da janela anterior, é necessário redefinir todos os contextos para que novas máscaras sejam geradas.

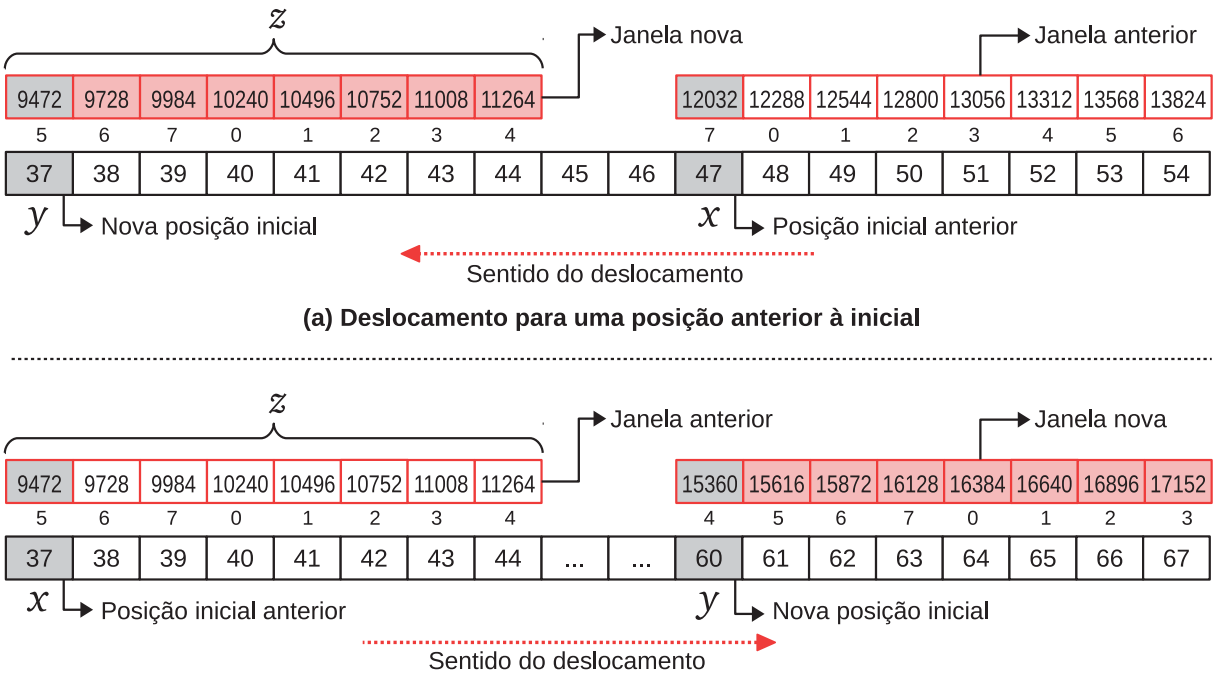


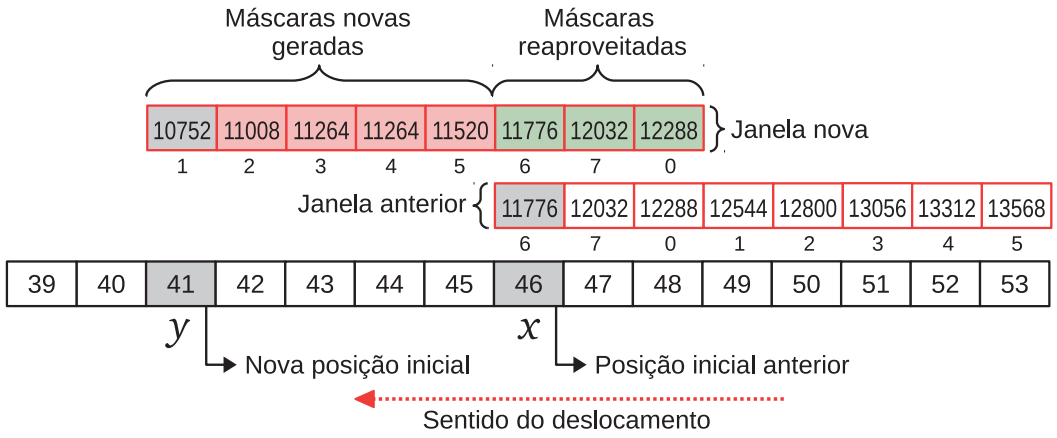
Figura 6.3: Deslocamento total da janela de contextos.

As outras duas situações referem-se àquelas onde, após o deslocamento da janela, parte da nova janela acaba se sobrepondo a janela anterior. São situações que ocorrem quando a janela é deslocada para uma posição anterior próxima, ou seja, $(x - y) <= z$; e quando é deslocada para uma posição posterior próxima, ou seja, $(y - x) <= z$. A Figura 6.4 (a) ilustra o primeiro caso e (b) o segundo. Nestas situações, a janela que ocupava a posição anterior contém algumas máscaras que podem ser reaproveitadas, não sendo necessário redefinir os contextos das posições que se sobrepõem, tornando o processo mais eficiente.

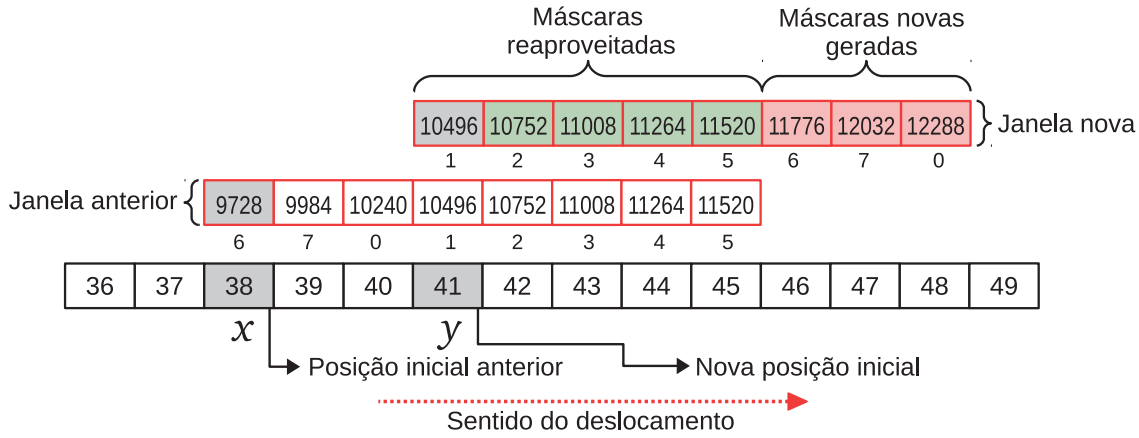
6.4 EncFS++

Foi necessário implementar algumas funções e métodos adicionais para adaptar o EncFS, operando no modo CTR implementado e descrito no capítulo anterior, a sua utilização com a WAESlib. As funções e métodos mais relevantes, descritos nesta seção, estão diretamente ligados à implementação e utilização na prática dos *pools* de contextos utilizados nos processos de cifragem e decifragem de blocos. Convencionamos chamar essa versão final que opera no modo CTR e capaz de realizar a cifragem especulativa em GPU de EncFS++.

Quando o SAC é montado, um dos primeiros métodos executados é o `EncFS_Context::initializeWAESEnvironment()`. Ele utiliza principalmente duas funções da WAESlib: `WAES_init()` e `WAES_setkey()`. Na primeira, é possível definir a quantidade total de contextos que serão criados pela biblioteca e disponibilizados à aplicação, bem com o tamanho dos segmentos utilizados. A chamada à função `WAES_setKey()` serve para configurar a chave de cifragem que será utilizada na função de definição de contextos.



(a) Deslocamento para uma posição anterior próxima



(b) Deslocamento para uma posição posterior próxima

Figura 6.4: Deslocamento parcial da janela de contextos.

Também é inicializada uma lista (`poolList`) com números sequenciais que identificam os *pools* de contextos. Ela é utilizada no controle da alocação dos *pools* dentro da aplicação.

Para o controle da alocação dos *pools*, foi implementado o método `EncFS_Context::getContextPool()`. Ele recebe como parâmetro a quantidade de contextos que o *pool* deve conter. Para permitir uma posterior concatenação do número lógico do *pool* e o número dos contextos contidos dentro do *pool*, é definida uma unidade de alocação. De acordo com a quantidade de contextos solicitados, um ou mais *pools* são alocados. Os *pools* alocados são removidos da lista `poolList`. O valor retornado do método é o identificador lógico do *pool*. Existe também o método `EncFS_Context::releaseContextPool()`, o qual é chamado no fechamento de arquivos com o objetivo de liberar os *pools* de contextos alocados na sua abertura.

Durante a inicialização do SAC, também é chamado o método `EncFS_Context::_initializeEncCtxPool()`. Ele é utilizado para criar e inicializar o *pool* de contextos utilizados na escrita. Após chamar o método `EncFS_Context::getContextPool()`, para alocar um *pool* de contextos, ele copia o valor atual do contador do SAC e chama a função `WAES_ctx()` para definir cada um dos contextos contidos no *pool*. A cada iteração, o valor do contador copiado é incrementado e utilizado como *nonce* na definição do próximo contexto. Este processo tem por objetivo

justamente disparar a geração inicial das máscaras de cifragem. Posteriormente, para saber qual contexto deve ser utilizado para recuperar a máscara gerada e realizar a cifragem, é armazenado o número de identificação do contexto inicial dentro do *pool*.

Os *pools* utilizados nos processos de leitura são criados e inicializados na abertura dos arquivos. Para tanto foi criado o método `FileNode::_initializeDecCtxPool()`. Ele chama o método `EncFS_Context::getContextPool()` para alocar um *pool* de contextos. Por padrão, os contextos desse *pool* são definidos para os blocos iniciais do arquivo e na quantidade correspondente ao tamanho do *pool*. Neste processo são recuperados os *nonces* dos blocos iniciais e utilizada a função `WAES_ctx()` de forma iterativa para disparar a geração das máscaras de decifragem utilizando cada um dos contextos contidos no *pool*. O principal objetivo do método `FileNode::_initializeDecCtxPool()` é justamente criar a janela de máscaras de decifragem, posicionando-a no início do arquivo.

Os parágrafos anteriores apresentaram as funções e métodos relacionados principalmente com a inicialização da biblioteca WAESlib e aquelas que lidam com a criação e preparação dos *pools* de escrita e leitura. Como esses *pools* são utilizados e ajustados durante os processos de leitura e escrita de blocos é exemplificado e discutido nas duas subseções seguintes.

6.4.1 Exemplo de utilização do *pool* de contextos para escrita

Quando o SAC é montado, o *pool* de contextos para escrita é criado, sendo chamada de forma iterativa a função `WAES_ctx()` para definir os contextos do *pool* e comandar a geração das máscaras de cifragem. A utilização dessas máscaras se dá nos processos de escrita de blocos e está ilustrado no diagrama de sequência da Figura 6.5. Até o chamado do método `CipherFileIOCTR::writeOneBlock()`, o procedimento ocorre de forma idêntica ao implementado no modo CTR, incluindo a execução dos métodos responsáveis pela alocação e carregamento de *nnodes* e grupos de *nonces*.

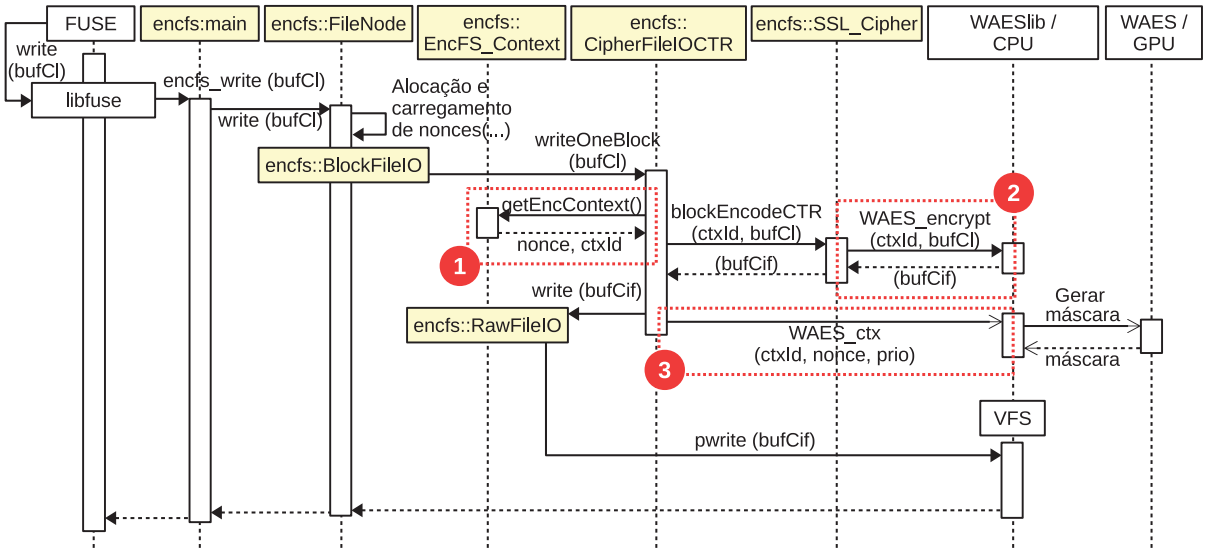


Figura 6.5: Processamento de uma requisição de `write()` com cifragem sendo realizada na GPU.

Quando a requisição de escrita de um bloco chega na classe `CipherFileIOCTR`, é chamado o método `EncFS_Context::getEncContext()`, indicado pelo número um na Figura 6.5. Ele faz uma cópia do contador atual do SAC e realiza o seu incremento. Esse valor copiado é um dos parâmetros de retorno do método, correspondendo ao *nonce* que foi utilizado

para gerar a máscara de cifragem. É necessário copiá-lo para as estruturas que armazenam os *nonces* dos blocos do arquivo para que futuramente possa ser recuperado e utilizado no processo de decifragem. Esse método também retorna um número de identificação de contexto. Esse número é formado pela concatenação do número de identificação lógico do *pool* e um número que indica o contexto dentro do *pool* correspondente ao início da fila. O número indicador do início da fila é então incrementado através de uma operação de adição modular. Esse esquema permite controlar o *pool* de contextos para escrita como uma fila circular virtual, onde as únicas informações necessárias para seu controle são o número de identificação lógico do *pool* e o número identificador do contexto dentro do *pool* correspondente ao início da fila.

O número de contexto retornado, em conjunto com o *buffer* contendo os dados do bloco a ser cifrado, são repassados ao método `SSL_Cipher::blockEncodeCTR()`. Esse por sua vez, chama a função `WAES_encrypt()` para realizar a cifragem do *buffer*, indicado pelo número dois na Figura 6.5. Como a geração das máscaras é disparada na montagem do SAC, quando a função `WAES_encrypt()` for chamada, serão grandes as chances da máscara de cifragem já ter sido gerada na GPU, transferida para a memória de sistema e estar prontamente disponível para utilização pela WAESlib. Através do número do contexto, a biblioteca WAESlib sabe qual máscara utilizar para realizar a operação de `XOR` com o *buffer* a ser cifrado.

Deste ponto em diante, no retorno dos métodos à classe `CipherFileIOCTR`, o *buffer* já se encontra cifrado. Antes de ser encaminhado à classe `RawFileIO` para ser gravado no SAB, é chamada a função `WAES_ctx()` para redefinir o contexto de cifragem recém utilizado, visando gerar uma nova máscara e manter o *pool* de contextos de escrita sempre com máscaras novas. Esta parte está indicada pelo número três na Figura 6.5. Neste caso, a prioridade escolhida será a menor possível, pois a máscara resultante estará no final da fila. O acionamento da função `WAES_ctx()` é assíncrono, bem como o funcionamento interno das funções da WAESlib que disparam a geração de novas máscaras na GPU e gerenciam a cópia das mesmas da memória da GPU para a memória de sistema.

6.4.2 Exemplo de utilização do *pool* de contextos para leitura

O diagrama da Figura 6.6 ilustra os principais passos envolvidos no tratamento de uma requisição de leitura. Assim com o exemplo da subseção anterior, até chegar na classe `CipherFileIOCTR`, os processos são idênticos aos implementados no modo CTR. Considerando que a janela de máscaras de decifragem é criada e posicionada no início do arquivo quando ele é aberto, caso a leitura inicie no bloco zero e mantenha um comportamento sequencial, o processo é mais simples. Neste caso, o bloco sendo acessado coincidirá com o início da janela e as etapas indicadas com o número um e dois no diagrama não serão executadas. O fluxo de execução continua na classe `RawFileIO` com o bloco cifrado sendo lido do disco e retornando à classe `CipherFileIOCTR`.

No passo três, é chamado o método `SSL_Cipher::blockDecodeCTR()`, repassando-se o *buffer* contendo o bloco cifrado. Como o bloco sendo acessado coincide com o início da janela, o contexto que contém a sua máscara necessária para decifragem também será o primeiro da janela. Então é chamada a função `WAES_decrypt()`. Como a janela foi populada com máscaras de decifragem na abertura do arquivo, são grandes as chances da máscara já terá sido produzida e estar disponível na WAESlib para ser utilizada. Após a aplicação da máscara sobre o *buffer*, a função então retorna com o *buffer* decifrado e o processo continua normalmente em direção à libfuse.

Na sequência, caso venha a requisição de leitura do próximo bloco, ou seja, o bloco de número um, o processo sofre uma ligeira modificação. No passo um, o início da ja-

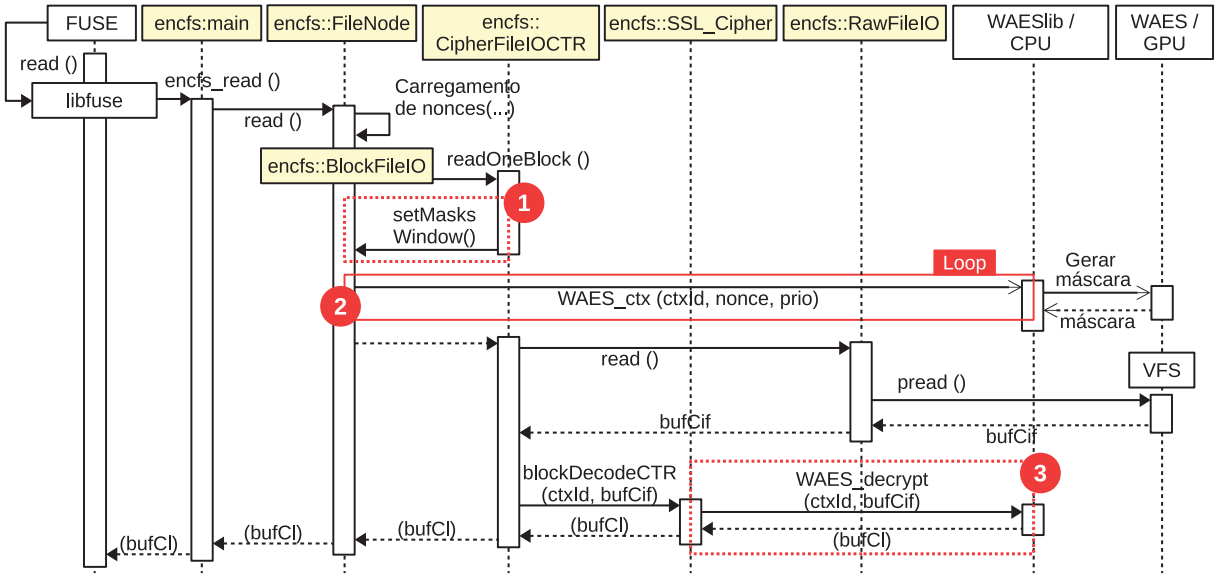


Figura 6.6: Processamento de uma requisição de `read()` com decifragem sendo realizada na GPU.

nela não coincidirá mais com o bloco sendo acessado, pois o mesmo ainda estará posicionado no contexto referente ao bloco acessado anteriormente. Então é chamado o método `FileNode::setMasksWindow()` para ajustar a janela. Como neste caso a diferença do início da janela em relação ao bloco sendo acessado é de apenas uma posição, o início da janela é ajustado para o bloco atual e o contexto contendo a máscara do bloco acessado anteriormente é redefino através da chamada à função `WAES_ctx()`. Essa redefinição de contexto tem por objetivo disparar a geração da máscara para o bloco *n* posições à frente do bloco atualmente sendo acessado, onde *n* corresponde ao tamanho da janela. Este é o comportamento padrão da implementação num processo de leitura sequencial.

Caso ocorra a leitura de um bloco não sequencial ao anteriormente lido, típico na situação de leitura aleatória, calcula-se a diferença do bloco atualmente sendo acessado e o número do bloco correspondente à posição inicial da janela de máscaras. Caso essa diferença seja maior que o tamanho da janela, as máscaras atualmente contidas nela não corresponderão aos blocos sendo acessados. Por isto, todos os contextos precisam ser redefinidos para gerarem novas máscaras. Isto é o que acontece no passo dois, onde o mesmo é repetido a quantidade de vezes correspondente ao tamanho da janela. Para cada contexto redefinido, a prioridade vai sendo gradativamente reduzida. Dessa forma, as máscaras dos blocos no início da janela ficam prontas antes.

Porém, dependendo do tamanho da janela, não é interessante simplesmente ir redefinindo os contextos diminuindo gradativamente a prioridade. A biblioteca WAESlib tem a capacidade de agregar contextos na hora de disparar a geração das máscaras, porém essa agregação acontece somente com contextos que possuem a mesma prioridade. Portanto, é interessante que somente os contextos referentes ao início da janela sejam definidos com um esquema de prioridades que vai diminuindo gradativamente, partindo da maior prioridade. Isto garante que as máscaras do início da janela fiquem prontas antes. Os contextos correspondentes ao restante da janela podem ser redefinidos usando uma valor igual de prioridade, permitindo que WAESlib realize sua agregação na hora de disparar a geração das máscaras.

Caso o bloco sendo acessado esteja situado a uma distância menor que o tamanho da janela, nem todos os contextos são redefinidos, pois ocorrerá a sobreposição entre a janela anterior

e a nova. A lógica contida no método `FileNode::setMasksWindow()` se encarrega de não redefinir os contextos que contenham máscaras que podem ser reaproveitadas, agilizando o processo e evitando o recômputo desnecessário de máscaras. Ao final do passo dois, o indicador do início da janela é atualizado.

Armazenando-se o valor correspondente ao número do bloco que indica o início da janela e realizando a redefinição dos contextos dentro da janela de forma circular, a estrutura de dados da janela, assim como o *pool* de contextos para escrita, também corresponde a uma fila circular virtual.

Com a janela ajustada, o processo continua normalmente como no acesso sequencial. É interessante notar que entre os passos um e três ocorre a leitura do bloco cifrado do SAB. Isto é útil pois, no caso de uma redefinição das posições iniciais da janela, enquanto o bloco é lido do SAB, as máscaras do início da janela vão sendo geradas em paralelo, aumentando as chances de já estarem prontas quando chegar no passo três.

Na próxima seção serão apresentados alguns resultados inicias que procuram demonstrar como ficou o desempenho do EncFS++ ao utilizar o processamento das funções de criptografia em GPU através da utilização do WAES e da biblioteca WAESlib, bem como explorando a utilização dos *pools* de contextos com a finalidade de realizar a cifragem especulativa nos processos de cifragem e decifragem de blocos.

6.5 Avaliação de vazão e utilização de CPU

Devido à extensão do trabalho, tempo dedicado à implementação do modo CTR e prazos, não foi possível realizar um conjunto de avaliações tão complexo quanto o realizado sobre a implementação do modo CTR. Por isso, para este capítulo, foram realizados testes de *microbenchmark* medindo apenas vazão e utilização de CPU. Seu principal objetivo foi demonstrar alguns resultados iniciais da utilização do WAES e da WAESlib, bem como validar a técnica de utilização dos *pools* de contextos anteriormente descritos.

O ambiente criado para realizar as medições é praticamente o mesmo descrito na Subseção 5.3.1 do Capítulo 5, com exceção dos tamanhos das requisições. Devido à alteração na configuração de hardware, várias medições realizadas no capítulo anterior precisaram ser refeitas para este capítulo. Visando agilizar a coleta de dados e simplificar a análise, este cenário se limitou a medir três tamanhos diferentes de requisição: 4, 64 e 128 KiB. Para a medição de utilização da CPU foi utilizada a ferramenta `pidstat`[3], monitorando exclusivamente o processo e subprocessos do EncFS.

Os testes foram realizados em um computador diferente do capítulo anterior pois o mesmo não permitia a instalação de uma GPU mais atual. Foi utilizado um notebook com o SO Linux e *kernel* 4.10.0, rodando em processador Intel Core i7-7700HQ a 2,8 GHz (frequência fixa e com oito processadores lógicos), 32 GiB de memória RAM DDR4 atuando em *dual channel* e disco de estado sólido (SSD) Western Digital modelo WDS240G1G0A com taxa de vazão teórica máxima para leitura por volta de 527.343 KiB/s (aproximadamente 514,98 MiB/s) e para gravação de 454.101 KiB/s (aproximadamente 443,46 MiB/s). A versão da biblioteca libfuse utilizada foi a 2.9.4, da biblioteca OpenSSL a 1.0.2g e da WAESlib a 2.01g0. A GPU utilizada foi uma NVIDIA GeForce GTX 1070 (versão para notebook)[4], em ambiente CUDA versão 9.2.

[3]Nas avaliações de desempenho deste capítulo a tecnologia *Hyper-Threading* não foi desativada, portanto os valores gerados pela ferramenta foram divididos pela quantidade total de processadores lógicos utilizados.

[4]Para evitar variações de desempenho significativas, a placa foi configurada para operar no modo de desempenho máximo, através da opção “GPUPowerMizerMode=1”.

Para avaliar as variações de vazão e utilização de CPU, foram comparadas diferentes versões do EncFS. A primeira, identificada pelo termo EncFS++, realizando processamento das funções criptográficas em GPU e a etapa final de `XOR` do modo CTR na CPU. Esta versão faz uso justamente do WAES e da biblioteca WAESlib. Como o WAES exige a utilização do modo CTR, nesta versão foi utilizado o modo CTR implementado no capítulo anterior. As outras duas versões são as com processamento exclusivo em CPU: uma utilizando o modo padrão do EncFS, o CBC, e a outra utilizando o mesmo modo CTR implementado no capítulo anterior e utilizado na versão EncFS++.

Considerando-se que o processador do equipamento onde foram realizadas as medições para este capítulo é mais atual e possui suporte a AES-NI[5], cada versão com processamento exclusivo em CPU foi dividida em duas: uma fazendo uso das instruções AES-NI e a outra não. A versão utilizando CBC e AES-NI foi denominada CBC(AESNI); sem utilizar AES-NI, CBC(S/AESNI). A versão utilizando CTR e AES-NI foi denominada CTR(AESNI); sem utilizar AES-NI, CTR(S/AESNI). Portanto, no total, foram realizadas medições utilizando-se cinco versões diferentes do EncFS: CBC(S/AESNI), CBC(AESNI), CTR(S/AESNI), CTR(AESNI) e EncFS++.

Como em alguns casos os valores de vazão alcançados foram limitados pela vazão suportada pelo disco, as cinco versões também foram analisadas em um cenário onde o SAB foi armazenado em memória, mais especificamente dentro do diretório `/run/shm`. Os valores obtidos nestes dois cenários (disco e memória) bem como uma análise dos resultados estão descritos nas subseções seguintes.

6.5.1 Resultados com o SAB armazenado em SSD

A Figura 6.7 (a) e (b) apresenta os valores de vazão alcançados pelas cinco versões do EncFS nas operações de leitura e escrita sequenciais, respectivamente.

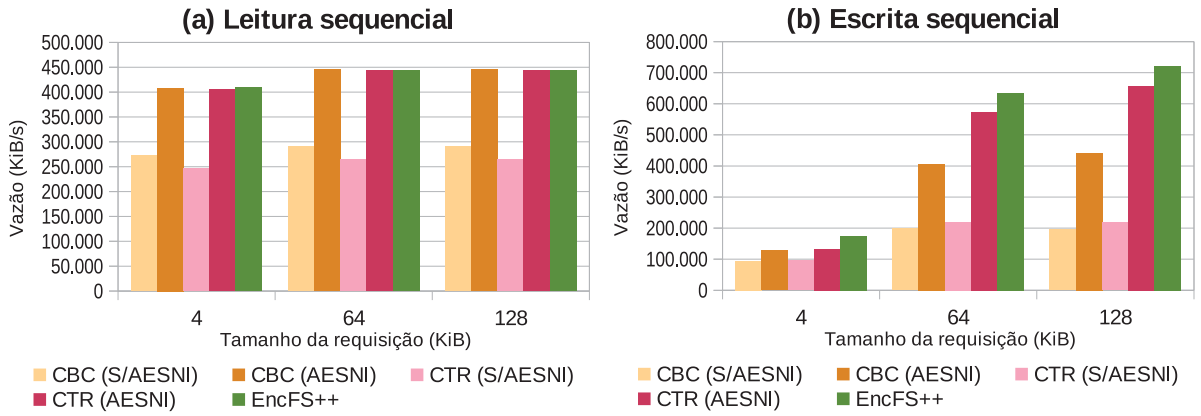


Figura 6.7: Níveis de vazão em (a) leitura sequencial e (b) escrita sequencial (SAB em disco).

As Tabelas 6.1 e 6.2 apresentam os valores de vazão e utilização de CPU para cada tamanho de requisição, bem como a média dos três tamanhos. Cada tabela apresenta os valores correspondentes às operações de leitura sequencial e escrita sequencial, respectivamente. Nestas tabelas são realizadas comparações entre a versão EncFS++ e as versões que utilizam o modo CBC e CTR com e sem AES-NI. Na comparação apresentada nas últimas quatro colunas de cada

[5]Sigla do inglês *Advanced Encryption Standard New Instructions*, uma denominação dada pela empresa Intel a um conjunto de instruções específicas para seus processadores que visa acelerar a execução de funções criptográficas que utilizam o cifrador AES [92].

tabela, pode-se verificar o ganho ou perda nos valores de vazão, bem como o índice de eficiência na utilização de CPU. A parte superior das tabelas compara a versão EncFS++ com a versão CBC; a parte inferior compara a versão EncFS++ com a versão CTR.

Tabela 6.1: Níveis de vazão e utilização de CPU em leitura sequencial (SAB em disco).

Leitura Sequencial (EncFS++ vs CBC)										
Tam. Req. (KiB)	CBC				EncFS++		EncFS++/CBC (S/AESNI)		EncFS++/CBC (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	271.947,00	11,32	408.063,80	10,54	409.819,40	14,71	50,70	1,1599	0,43	0,7199
64	290.076,90	12,10	444.789,00	5,69	443.007,10	11,54	52,72	1,6021	-0,40	0,4914
128	290.709,90	12,11	444.627,40	5,48	443.559,20	11,54	52,58	1,6003	-0,24	0,4733
Média	**284.244,60**	**11,84**	**432.493,40**	**7,24**	**432.128,57**	**12,60**	52,03	1,4294	-0,08	0,5741
Leitura Sequencial (EncFS++ vs CTR)										
Tam. Req. (KiB)	CTR				EncFS++		EncFS++/CTR (S/AESNI)		EncFS++/CTR (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	246.701,90	11,26	404.463,70	10,42	409.819,40	14,71	66,12	1,2712	1,32	0,7177
64	264.562,10	12,08	443.480,40	5,57	443.007,10	11,54	67,45	1,7532	-0,11	0,4823
128	264.122,80	12,08	443.436,20	5,18	443.559,20	11,54	67,94	1,7571	0,03	0,4489
Média	**258.462,27**	**11,80**	**430.460,10**	**7,06**	**432.128,57**	**12,60**	67,19	1,5668	0,39	0,5624

Tabela 6.2: Níveis de vazão e utilização de CPU em escrita sequencial (SAB em disco).

Escrita Sequencial (EncFS++ vs CBC)										
Tam. Req. (KiB)	CBC				EncFS++		EncFS++/CBC (S/AESNI)		EncFS++/CBC (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	95.146,80	10,41	127.551,00	9,79	175.647,40	11,75	84,61	1,6356	37,71	1,1467
64	199.012,10	10,55	405.461,60	8,37	635.454,40	14,16	219,30	2,3784	56,72	0,9257
128	195.190,30	9,93	442.628,50	8,30	721.119,30	11,75	269,44	3,1210	62,92	1,1503
Média	**163.116,40**	**10,30**	**325.213,70**	**8,82**	**510.740,37**	**12,56**	213,11	2,5678	57,05	1,1028
Escrita Sequencial (EncFS++ vs CTR)										
Tam. Req. (KiB)	CTR				EncFS++		EncFS++/CTR (S/AESNI)		EncFS++/CTR (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	95.702,20	10,37	133.028,10	9,66	175.647,40	11,75	83,54	1,6186	32,04	1,0854
64	219.947,20	10,35	573.651,60	7,43	635.454,40	14,16	188,91	2,1108	10,77	0,5809
128	217.895,90	9,74	657.295,00	6,30	721.119,30	11,75	230,95	2,7428	9,71	0,5878
Média	**177.848,43**	**10,15**	**454.658,23**	**7,80**	**510.740,37**	**12,56**	187,18	2,3217	12,34	0,6974

Os valores apresentados nas últimas quatro colunas foram coloridos da seguinte forma: para a coluna vazão, os números foram coloridos em verde, quando acima de zero, indicando ganho de vazão; em vermelho, quando abaixo de zero, indicando perda de vazão. Para a coluna eficiência de uso da CPU, os números foram coloridos em verde, para valores superiores a um, indicando aumento no nível de eficiência e, em vermelho, para valores abaixo de um, indicando diminuição.

O índice de eficiência na utilização da CPU procurou medir a variação na proporção de carga de trabalho realizado a cada um por cento de utilização da CPU das versões comparadas. A fórmula utilizada foi: $x = (y/z)/(w/k)$, onde x corresponde ao índice de eficiência sendo calculado, y e z correspondem ao valor de vazão e utilização de CPU alcançados na versão sendo

comparada; *w* e *k* correspondem ao valor de vazão e utilização de CPU alcançados na versão com a qual se está fazendo a comparação, respectivamente.

Avaliando-se a leitura sequencial, na comparação entre a versão EncFS++ e CBC(S/AESNI), há um aumento médio de 52,03% nos valores de vazão. Em relação à eficiência de uso da CPU, um aumento médio de 42,94%. Ao se comparar com a versão CBC(AESNI), praticamente não há variação na vazão, com uma redução média de apenas 0,08%, porém com uma redução significativa de 42,59% na eficiência de uso da CPU. Na comparação da versão EncFS++ com a versão CTR(S/AESNI), há um ganho médio de 67,19% na vazão e aumento de 56,68% na eficiência de uso da CPU. Comparando-se com a versão CTR(AESNI), também praticamente não há alteração na vazão, com um aumento médio de apenas 0,39% na vazão, mas também com diminuição significativa de 43,76% na eficiência de uso da CPU.

Nas comparações com as versões que não utilizam AES-NI, pode-se perceber que os ganhos na vazão ao se realizar o processamento em GPU são mais significativos, bem como os aumentos na eficiência de uso da CPU. Neste cenário de leitura, que envolve a decifragem de dados, o aumento significativo nos valores de vazão merece destaque, já que o CBC também pode ser processado de forma paralela e teoricamente não haveria grandes vantagens neste quesito em relação ao modo CTR. Isto pode ser comprovado comparando-se os valores médios de vazão entre as versões CBC(S/AESNI) e CTR(S/AESNI), bem como as versões CBC(AESNI) e CTR(AESNI): nestes casos, a simples utilização do modo CTR não trouxe nenhum ganho, pelo contrário, houve uma pequena perda.

Analisando-se o cenário que envolve a utilização do AES-NI, percebe-se o quanto a aceleração em hardware oferecida pelo próprio processador melhora significativamente os resultados, tanto com relação a obter valores maiores de vazão, quanto eficiência na utilização do processador. Neste caso, a versão EncFS++ apenas consegue manter os mesmos níveis de vazão. Porém, o mesmo não acontece ao se analisar a eficiência de uso da CPU, pois na versão EncFS++, mesmo com o processamento das funções criptográficas em GPU, acaba ocorrendo aumento na utilização de CPU.

É importante destacar que neste cenário de leitura sequencial com o SAB armazenado em disco, a análise dos ganhos nos valores de vazão fica comprometida. As versões que conseguem atingir taxas maiores de vazão tem valores próximos do limite teórico máximo de vazão suportado pelo disco, impedindo que diferenças maiores de vazão possam se manifestar. Como consequência, na Figura 6.8 (a), pode-se perceber que as versões CBC e CTR com AES-NI, bem como EncFS++, acabam ficando com valores muito próximos. Quando a limitação da velocidade de acesso em disco é eliminada, perceber-se melhor os ganhos da versão EncFS++, o que pode ser constatado na Figura 6.9 (a), a qual será discutida na próxima subseção.

Em compensação, na escrita sequencial, é possível perceber valores mais significativos de ganho na vazão. Na comparação da versão EncFS++ com a versão CBC(S/AESNI), o ganho de vazão em média chega a 213,11%, com aumento na eficiência de uso da CPU de 156,78%. Na comparação com a versão CTR(S/AESNI), o ganho médio de vazão é de 187,18% e aumento médio da eficiência de utilização da CPU de 132,17%.

Neste cenário, a possibilidade de paralelização na cifragem do modo CTR aliada ao processamento em GPU, resultam em ganho de vazão até mesmo sobre as versões que utilizam AES-NI. Comparando-se com a versão CBC(AESNI), o ganho médio de vazão chega a 57,05%; comparando-se com a versão CTR(AESNI), o ganho médio de vazão é de 12,34%. Com relação à eficiência de utilização da CPU, comparado à versão CBC(AESNI), há um aumento na eficiência de 10,28%. Em compensação, comparado à versão CTR(AESNI), há uma redução de 30,26%.

Os ganhos maiores de vazão apresentados neste cenário de escrita também se devem ao fato de que ela é realizada primeiramente em páginas de memória *cache*, para somente depois

serem gravadas em disco. Isto ajuda a eliminar o limite de vazão imposto pelo disco, permitindo que ganhos maiores de vazão se manifestem.

A Figura 6.8 (a) e (b) apresenta os valores de vazão para as operações de leitura e escrita aleatórias, respectivamente. Na sequência, as Tabelas 6.3 e 6.4 apresentam os valores de vazão e utilização de CPU relacionados a estas operações.

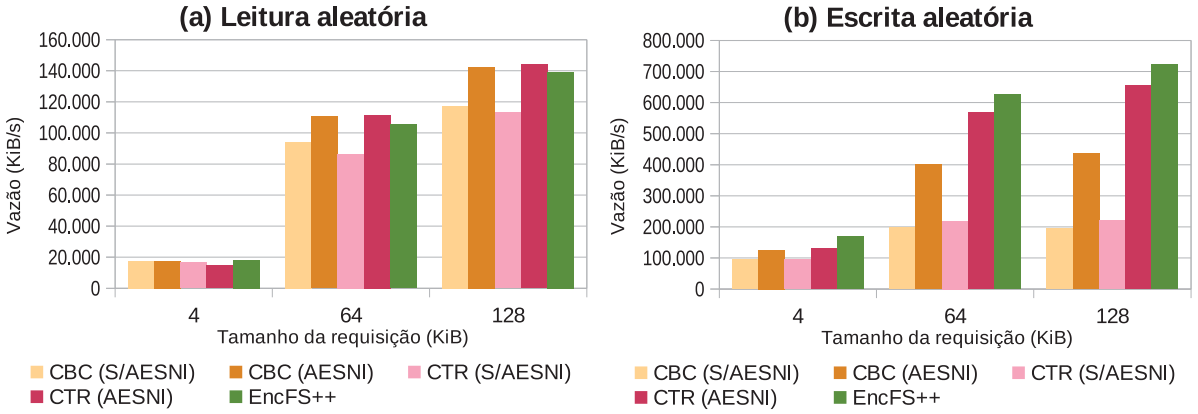


Figura 6.8: Níveis de vazão em (a) leitura aleatória e (b) escrita aleatória (SAB em disco).

Tabela 6.3: Níveis de vazão e utilização de CPU em leitura aleatória (SAB em disco).

Leitura Aleatória (EncFS++ vs CBC)										
Tam. Req. (KiB)	CBC				EncFS++		EncFS++/CBC (S/AESNI)		EncFS++/CBC (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	17.381,99	1,93	17.336,30	1,35	18.274,90	3,75	5,14	0,5414	5,41	0,3802
64	94.221,50	4,37	110.513,80	1,92	105.415,50	5,35	11,88	0,9139	-4,61	0,3423
128	117.357,70	5,34	142.500,70	2,19	139.155,80	5,38	18,57	1,1756	-2,35	0,3982
Média	**76.320,40**	**3,88**	**90.116,93**	**1,82**	**87.615,40**	**4,83**	14,80	0,9226	-2,78	0,3670
Leitura Aleatória (EncFS++ vs CTR)										
Tam. Req. (KiB)	CTR				EncFS++		EncFS++/CTR (S/AESNI)		EncFS++/CTR (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	16.488,30	1,87	14.675,50	1,26	18.274,90	3,75	10,84	0,5524	24,53	0,4192
64	86.588,60	4,95	111.722,60	1,88	105.415,50	5,35	21,74	1,1260	-5,65	0,3323
128	113.157,60	5,44	143.972,80	2,15	139.155,80	5,38	22,98	1,2435	-3,35	0,3860
Média	**72.078,17**	**4,09**	**90.123,63**	**1,76**	**87.615,40**	**4,83**	21,56	1,0291	-2,78	0,3555

Na leitura aleatória, os ganhos de vazão da versão EncFS++ sobre as versões sem AES-NI são mais modestos. Comparado com a versão CBC(S/AESNI), ficam em média por volta de 14,80% e comparado com a versão CTR(S/AESNI), 21,56%. Com relação à eficiência de utilização da CPU, comparado à versão CBC(S/AESNI) há uma redução de 7,74% e comparado à versão CTR(S/AESNI) um aumento de 2,91%.

Na comparação com ambas as versões que utilizam AES-NI, no quesito vazão, há uma redução média de 2,78%. Com relação à eficiência de uso da CPU, há reduções bem significativas: 63,30% na comparação com a versão CBC(AESNI) e 64,45% na comparação com a versão CTR(AESNI).

Para entender melhor os resultados obtidos neste cenário, é importante descrever como é o acesso aleatório gerado pela ferramenta `fio`. Nas requisições de 4 KiB, cada requisição

Tabela 6.4: Níveis de vazão e utilização de CPU em escrita aleatória (SAB em disco).

Escrita Aleatória (EncFS++ vs CBC)										
Tam. Req. (KiB)	CBC				EncFS++		EncFS++/CBC (S/AESNI)		EncFS++/CBC (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	94.346,50	10,36	125.473,70	9,64	169.392,70	11,54	79,54	1,6114	35,00	1,1277
64	197.781,90	10,50	399.925,60	8,64	625.814,20	14,06	216,42	2,3630	56,48	0,9615
128	194.243,90	9,91	437.508,90	8,32	721.976,20	11,60	271,69	3,1745	65,02	1,1831
Média	**162.124,10**	**10,26**	**320.969,40**	**8,87**	**505.727,70**	**12,40**	211,94	2,5798	57,56	1,1265
Escrita Aleatória (EncFS++ vs CTR)										
Tam. Req. (KiB)	CTR				EncFS++		EncFS++/CTR (S/AESNI)		EncFS++/CTR (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	95.020,80	10,32	131.463,40	9,55	169.392,70	11,54	78,27	1,5941	28,85	1,0660
64	218.925,00	10,32	568.135,70	7,46	625.814,20	14,06	185,86	2,0978	10,15	0,5841
128	220.347,30	9,87	656.317,20	6,70	721.976,20	11,60	227,65	2,7885	10,00	0,6353
Média	**178.097,70**	**10,17**	**451.972,10**	**7,90**	**505.727,70**	**12,40**	183,96	2,3289	11,89	0,7129

gerada pela ferramenta resultará na leitura de um bloco do SAC. Por sua vez, requisições de 64 KiB resultam no acesso a 16 blocos e 128 KiB a 32 blocos. A escolha do *offset* dentro do arquivo no qual ocorrerá a leitura do primeiro bloco da requisição é aleatória e baseada numa distribuição uniforme. Consequentemente, cada requisição resulta sempre no acesso a blocos diferentes, não ocorrendo a repetição na leitura dos blocos.

Neste cenário, a simples utilização do *pool* de contextos para leitura e, consequentemente da janela de máscaras, com um tamanho fixo não seria eficiente. A cada nova requisição gerada, a probabilidade da janela ser totalmente deslocada é alta. Isto faz com que seja necessário disparar a geração de novos contextos na quantidade correspondente ao tamanho da janela, visando produzir novas máscaras. Dependendo do tamanho da janela e do tamanho da requisição, isto pode ocasionar uma sobrecarga considerável, tanto no sentido de se disparar a produção de novas máscaras, quanto no sentido de se produzir máscaras que provavelmente não serão utilizadas.

Portanto, para este cenário foram utilizados três tamanhos diferentes de janela: tamanho igual a dois para requisições de 4 KiB; igual a oito para 64 KiB e 16 para 128 KiB. Reduzir o tamanho da janela tem como objetivo principal evitar a produção de máscaras que não serão utilizadas, contudo, possui o efeito adverso de não permitir que as máscaras sejam geradas com muita antecedência. Com isto, o acionamento da função `WAES_decrypt()` pode bloquear devido ao fato da máscara ainda não estar pronta. Este efeito adverso pode ser observado nos resultados obtidos para as requisições de 64 e 128 KiB, onde ocorrem perdas na vazão.

Outra questão que também afetou os resultados apresentados para as requisições de 64 e 128 KiB é o mecanismo de deslizamento da janela conforme as máscaras vão sendo consumidas. Enquanto se está decifrando os blocos iniciais da requisição não há problemas, pois a cada máscara consumida uma nova é gerada para o bloco n posições à frente, onde n corresponde ao tamanho da janela. Considerando, por exemplo, requisições de 128 KiB e uma janela de 16 posições, até o processamento do 15º bloco da requisição, todas as máscaras produzidas serão aproveitadas. As máscaras produzidas no processamento do 16º bloco ao 31º não serão aproveitadas, pois a próxima requisição muito provavelmente não lerá os blocos subsequentes ao 31º bloco. Assim, mesmo com a utilização de janelas menores, ocorre o disparo e a produção de máscaras que acabam não sendo utilizadas.

No caso do tratamento das requisições de 4 KiB, ao se utilizar uma janela de apenas duas posições, esse disparo e produção desnecessária de máscaras é reduzido consideravelmente. Isto ajuda a explicar os resultados positivos obtidos justamente neste caso. Comparando com a versão CBC(AESNI), há aumento de 5,41% na vazão e na comparação com a versão CTR(AESNI), o aumento chega a ser de 24,53%. Também contribui para esses resultados o fato de haver a latência de acesso a disco e o tempo de processamento entre as diferentes requisições da libfuse que permitem esconder a latência de produção das máscaras.

O cenário da escrita aleatória não traz muitas diferenças em relação à escrita sequencial, apresentando níveis de ganhos semelhantes. Isto é esperado devido ao fato da mesma ocorrer nas páginas de memória *cache* e também por utilizar o mesmo *pool* de contextos utilizado para a escrita sequencial. O ganho médio de vazão na comparação com a versão CBC(S/AESNI) é de 211,94% e na comparação com CTR(S/AESNI) é de 183,96%. Na comparação com a versão CBC(AESNI) o ganho de vazão é de 57,56% e com a versão CTR(AESNI) é de 11,89%. Na eficiência de utilização da CPU, há um aumento médio de 157,98% comparando-se com a versão CBC(S/AESNI) e 132,89% comparando-se com a versão CTR(S/AESNI). Comparando-se com a versão CBC(AESNI), há um aumento de 12,65% e com a versão CTR(AESNI), uma redução de 28,71%.

6.5.2 Resultados com o SAB armazenado em memória

A Figura 6.9 (a) e (b) apresenta os valores de vazão alcançados nas operações de leitura e escrita sequencial, respectivamente, com o SAB armazenado em memória. Na sequência, as Tabelas 6.5 e 6.6 apresentam os valores correspondentes.

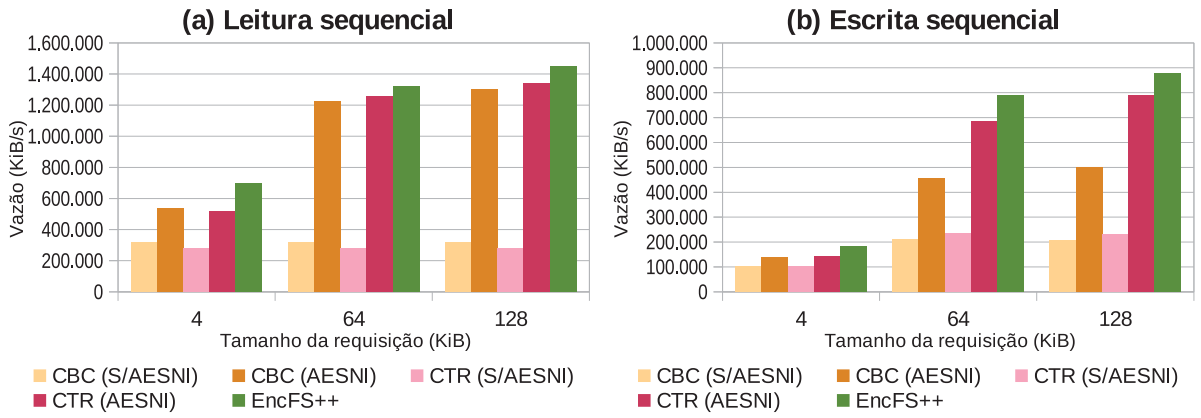


Figura 6.9: Níveis de vazão em (a) leitura sequencial e (b) escrita sequencial (SAB em memória).

O cenário com o SAB armazenado em memória confirma que ainda há margem para ganhos maiores de vazão ao se remover o limite de vazão imposto pelo disco. Na comparação com as versões sem AES-NI, há ganhos bastante significativos. Comparado com a versão CBC(S/AESNI), há aumento médio de 266,86% na vazão, em relação à versão CTR(S/AESNI), 311,38%. O aumento na eficiência de uso da CPU chegou a 93,02% na comparação com a versão CBC(S/AESNI) e 116,38% na comparação com a versão CTR(S/AESNI).

Comparando-se com as versões que utilizam AES-NI, também há aumento nos valores de vazão, porém menores. Na comparação com a versão CBC(AESNI), esse ganho atinge em média 13,66% e com a versão CTR(AESNI), 11,55%. Contudo, há reduções significativas na eficiência de uso da CPU. Na comparação com a versão CBC(AESNI), essa redução foi de 40,33% e com a versão CTR(AESNI), 41,53%.

Tabela 6.5: Níveis de vazão e utilização de CPU em leitura sequencial (SAB em memória).

Leitura Sequencial (EncFS++ vs CBC)										
Tam. Req. (KiB)	CBC				EncFS++		EncFS++/CBC (S/AESNI)		EncFS++/CBC (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	315.673,40	12,20	532.391,00	12,20	698.068,30	21,76	121,14	1,2396	31,12	0,7350
64	315.344,20	12,20	1.223.185,60	12,17	1.323.435,30	23,90	319,68	2,1427	8,20	0,5510
128	315.456,70	12,20	1.299.305,90	12,15	1.450.770,10	23,90	359,90	2,3472	11,66	0,5676
Média	**315.491,43**	**12,20**	**1.018.294,17**	**12,17**	**1.157.424,57**	**23,19**	266,86	1,9302	13,66	0,5967
Leitura Sequencial (EncFS++ vs CTR)										
Tam. Req. (KiB)	CTR				EncFS++		EncFS++/CTR (S/AESNI)		EncFS++/CTR (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	281.280,50	12,19	516.515,50	12,20	698.068,30	21,76	148,18	1,3907	35,15	0,7575
64	281.257,30	12,20	1.255.543,00	12,15	1.323.435,30	23,90	370,54	2,4015	5,41	0,5357
128	281.521,80	12,20	1.340.740,80	12,12	1.450.770,10	23,90	415,33	2,6295	8,21	0,5486
Média	**281.353,20**	**12,20**	**1.037.599,77**	**12,15**	**1.157.424,57**	**23,19**	311,38	2,1638	11,55	0,5847

Tabela 6.6: Níveis de vazão e utilização de CPU em escrita sequencial (SAB em memória).

Escrita Sequencial (EncFS++ vs CBC)										
Tam. Req. (KiB)	CBC				EncFS++		EncFS++/CBC (S/AESNI)		EncFS++/CBC (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	100.473,60	10,42	136.500,80	9,82	183.700,80	12,19	82,83	1,5625	34,58	1,0847
64	209.676,50	11,00	455.016,00	9,66	788.011,50	18,59	275,82	2,2234	73,18	0,9004
128	206.597,40	10,43	498.829,30	9,66	876.223,20	18,94	324,12	2,3349	75,66	0,8959
Média	**172.249,17**	**10,61**	**363.448,70**	**9,72**	**615.978,50**	**16,57**	257,61	2,2902	69,48	0,9936
Escrita Sequencial (EncFS++ vs CTR)										
Tam. Req. (KiB)	CTR				EncFS++		EncFS++/CTR (S/AESNI)		EncFS++/CTR (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	100.370,90	10,45	142.324,10	9,78	183.700,80	12,19	83,02	1,5691	29,07	1,0357
64	234.054,10	10,89	685.164,60	8,49	788.011,50	18,59	236,68	1,9729	15,01	0,5251
128	231.036,10	10,24	791.013,20	8,26	876.223,20	18,94	279,26	2,0504	10,77	0,4828
Média	**188.487,03**	**10,53**	**539.500,63**	**8,84**	**615.978,50**	**16,57**	226,80	2,0759	14,18	0,6090

O destaque para este cenário, além dos aumentos significativos de vazão alcançados, é o bom desempenho da versão EncFS++ mesmo considerando que o SAB está armazenado em memória. A princípio isto poderia ser um problema, pois ao se eliminar a latência de acesso a disco, fica mais difícil esconder a latência associada à geração das máscaras. No entanto, a utilização do *pool* de contextos para leitura com um tamanho razoável (64 contextos), atuando como uma janela deslizante, permite gerar as máscaras com a antecedência adequada. O aumento nos níveis de vazão ajudam a comprovar a eficiência de sua utilização, pois se as máscaras não ficassem prontas no tempo adequado, a aplicação bloquearia na hora de aplicar as máscaras, consequentemente aumentando o tempo de decifragem e reduzindo os níveis de vazão.

Os resultados obtidos na escrita sequencial são semelhantes aos obtidos na escrita sequencial com o SAB armazenado em disco, pois a escrita também ocorre em memória. Neste cenário, há apenas um ligeiro aumento nos valores obtidos. Comparando-se com a versão CBC(S/AESNI), obtém-se aumento médio de 257,61% na vazão e com a versão CTR(S/AESNI), 226,80%. Há aumento médio na eficiência de utilização da CPU de 129,02% ao se comparar com

a versão CBC(S/AESNI) e 107,59% com a versão CTR(S/AESNI). Na comparação com a versão CBC(AESNI), praticamente não há variação na eficiência de uso da CPU. Já na comparação com a versão CTR(AESNI) há redução de 39,10%.

A Figura 6.10 (a) e (b) apresenta os resultados de vazão alcançados nas operações de leitura e escrita aleatórias. Na sequência, as Tabelas 6.7 e 6.8 apresentam os valores correspondentes de vazão e utilização de CPU.

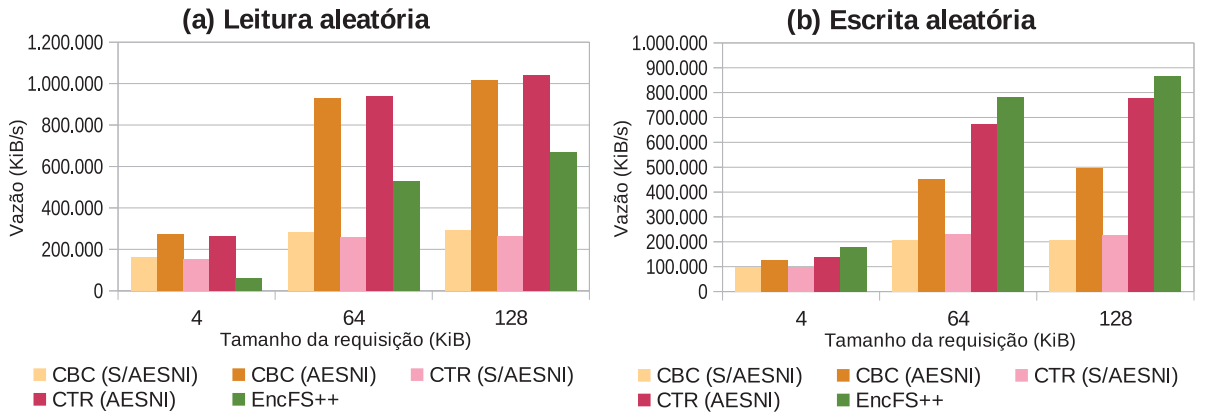


Figura 6.10: Níveis de vazão em (a) leitura aleatória e (b) escrita aleatória (SAB em memória).

Tabela 6.7: Níveis de vazão e utilização de CPU em leitura aleatória (SAB em memória).

Leitura Aleatória (EncFS++ vs CBC)										
Tam. Req. (KiB)	CBC				EncFS++		EncFS++/CBC (S/AESNI)		EncFS++/CBC (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	162.777,60	10,37	274.526,80	9,15	58.321,20	8,30	-64,17	0,4476	-78,76	0,2342
64	283.924,40	11,40	928.901,00	9,80	528.848,20	17,12	86,26	1,2402	-43,07	0,3260
128	290.148,60	11,45	1.016.858,50	9,89	668.070,60	17,37	130,25	1,5176	-34,30	0,3738
Média	**245.616,87**	**11,07**	**740.095,43**	**9,61**	**418.413,33**	**14,26**	70,35	1,3224	-43,46	0,3810
Leitura Aleatória (EncFS++ vs CTR)										
Tam. Req. (KiB)	CTR				EncFS++		EncFS++/CTR (S/AESNI)		EncFS++/CTR (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	151.669,50	10,56	264.138,30	9,33	58.321,20	8,30	-61,55	0,4891	-77,92	0,2481
64	255.632,50	11,45	938.167,50	9,81	528.848,20	17,12	106,88	1,3841	-43,63	0,3229
128	262.777,10	11,52	1.042.024,20	9,83	668.070,60	17,37	154,23	1,6864	-35,89	0,3628
Média	**223.359,70**	**11,18**	**748.110,00**	**9,66**	**418.413,33**	**14,26**	87,33	1,4681	-44,07	0,3786

Comparando-se com a versão CBC(S/AESNI), há um ganho médio de 70,35% na vazão, porém com uma perda considerável de 64,17% nas requisições de 4 KiB. Comparando-se com a versão CTR(S/AESNI), há um ganho médio de 87,33%, também com perda significativa de 61,55% nas requisições de 4 KiB. Analisando-se a eficiência de utilização da CPU, comparado à versão CBC(S/AESNI), há um aumento médio na eficiência de 32,24%. Comparado à versão CTR(S/AESNI), o aumento médio é de 46,81%. Na comparação com ambas as versões, no caso das requisições de 4 KiB, há uma redução de eficiência por volta de 55,24% comparado à versão CBC(S/AESNI) e 51,09% na comparação com a versão CTR(S/AESNI).

Os resultados que comparam a versão EncFS++ com as versões que utilizam AES-NI apresentam perdas significativas de vazão. Na comparação com ambas as versões, CBC(AESNI)

Tabela 6.8: Níveis de vazão e utilização de CPU em escrita aleatória (SAB em memória).

Escrita Aleatória (EncFS++ vs CBC)										
Tam. Req. (KiB)	CBC				EncFS++		EncFS++/CBC (S/AESNI)		EncFS++/CBC (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	97.472,20	10,28	128.172,60	9,75	177.859,80	11,94	82,47	1,5714	38,77	1,1325
64	208.820,60	10,97	450.610,90	9,64	782.998,80	18,57	274,96	2,2143	73,76	0,9020
128	206.515,40	10,43	495.720,70	9,64	867.641,60	18,89	320,13	2,3189	75,03	0,8932
Média	**170.936,07**	**10,56**	**358.168,07**	**9,68**	**609.500,07**	**16,47**	256,57	2,2863	70,17	0,9998
Escrita Aleatória (EncFS++ vs CTR)										
Tam. Req. (KiB)	CTR				EncFS++		EncFS++/CTR (S/AESNI)		EncFS++/CTR (AESNI)	
	Sem AESNI		Com AESNI							
	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão (KiB/s)	CPU (%)	Vazão(%)	Eficiência uso CPU	Vazão(%)	Eficiência uso CPU
4	96.735,50	10,32	137.356,40	9,60	177.859,80	11,94	83,86	1,5884	29,49	1,0413
64	232.653,90	10,87	674.399,30	8,46	782.998,80	18,57	236,55	1,9690	16,10	0,5288
128	228.019,60	10,13	777.447,00	8,21	867.641,60	18,89	280,51	2,0402	11,60	0,4848
Média	**185.803,00**	**10,44**	**529.734,23**	**8,76**	**609.500,07**	**16,47**	228,04	2,0790	15,06	0,6117

e CTR(AESNI), a perda média é de aproximadamente 44% na vazão e a redução na eficiência de uso da CPU é de aproximadamente 62%. Os resultados obtidos neste cenário ajudam a visualizar a importância da geração antecipada das máscaras de decifragem e de se ocultar a latência envolvida na sua geração.

Conforme descrito no cenário de leitura aleatória em disco, as janelas foram reduzidas de acordo com o tamanho das requisições. No caso das requisições de 4 KiB, esse tamanho é de apenas dois contextos. Considerando esse tamanho e a característica do acesso aleatório, isto elimina a possibilidade de gerar as máscaras de forma antecipada. Quando a requisição chega, o disparo da geração da máscara precisa ser feito no mesmo instante. Como não há a latência do acesso a disco para ler o bloco inicialmente, já que a leitura é feita em memória, praticamente não há tempo no qual a máscara possa ser gerada de forma paralela à leitura. Consequentemente, quando chegar no ponto em que o bloco precisa ser decifrado, a máscara ainda não estará pronta, bloqueando a aplicação até que a mesma seja gerada na GPU e transferida para a memória de sistema. Isto ajuda a explicar porque as maiores perdas na vazão ocorrem justamente no tratamento das requisições de 4 KiB.

As perdas nos valores de vazão sofrem uma ligeira redução no caso do tratamento de requisições maiores, as de 64 e 128 KiB. Como a janela é um pouco maior, oito contextos no caso de 64 KiB e 16 contextos no caso de 128 KiB, ainda é possível gerar algumas máscaras com certa antecedência. Porém, como também não há a latência do acesso a disco e a janela precisa ser mantida pequena para evitar a geração desnecessária de máscaras, essa antecedência não é suficiente para deixar todas as máscaras prontamente disponíveis. Como consequência, o tratamento dessas requisições também sofre impacto ao ter que esperar as máscaras ficarem prontas.

No caso da escrita aleatória, o gráfico e a tabela foram acrescentados apenas por questões de completude. Como este cenário envolve a escrita de dados em memória e utilizam o mesmo *pool* de contextos, não há diferenças significativas de desempenho com relação aos resultados obtidos na escrita sequencial.

Tanto para o cenário analisado nesta subseção quanto na subseção anterior, há tabelas contendo os valores mínimo, máximo, desvio padrão e média tanto para vazão quanto para utilização de CPU. Essas tabelas podem ser consultadas no Apêndice C. Para analisar os limites máximos práticos de vazão possíveis de serem alcançados, também foi realizada uma análise

de desempenho onde o EncFS foi configurado para utilizar um cifrador mínimo e um nulo. No cifrador mínimo, uma simples operação de inversão total dos bits era aplicada sobre os dados contidos nos blocos. Ao se utilizar o cifrador nulo, os dados não sofriam alteração. No total foram comparadas três versões do EncFS: a versão EncFS++, utilizando o modo CTR e a geração das máscaras de cifragem em GPU; duas versões usando somente o modo CTR, porém com os cifradores mínimo e nulo, consequentemente não envolvendo o processamento em GPU. Os resultados podem ser vistos no apêndice D.

6.6 Considerações finais

Os resultados apresentados neste capítulo demonstram que a utilização da cifragem especulativa pode melhorar ainda mais o desempenho do modo CTR. Comprovamos esse ponto experimentalmente com o processamento em GPU das funções criptográficas, através da utilização do *kernel* WAES, presente em uma versão mais recente da biblioteca WAESlib. A utilização da cifragem especulativa CTR ajudou a aumentar os valores de vazão do EncFS em vários cenários. Os valores mais significativos foram obtidos com o SAB armazenado em memória. Por este motivo, a sumarização seguinte dos resultados diz respeito a este cenário.

Os maiores ganhos se manifestaram nos cenários nos quais não se fez uso das instruções AES-NI. Ao se analisar os valores de vazão, comparando-se com a versão CBC(S/AESNI), o ganho médio nas operações de leitura sequencial, escrita sequencial e escrita aleatória foi de 260,35%. O ganho médio na operação de leitura aleatória foi de 70,35%. Comparando-se com a versão CTR(S/AESNI), o ganho médio nas operações de leitura sequencial, escrita sequencial e escrita aleatória foi de 255,41%. O ganho médio na leitura aleatória foi de 87,33%.

Com relação a eficiência de utilização da CPU, conforme esperado, neste cenário houve aumento nos níveis de eficiência, o que na prática resulta em uma menor utilização de CPU considerando os níveis de vazão alcançados. Comparado à versão CBC(S/AESNI), o aumento médio da eficiência, considerando as quatro operações, foi de 95,73%. Já na comparação com a versão CTR(S/AESNI), o aumento médio foi de 94,67%.

Nos cenários onde o processamento das funções criptográficas em CPU faz uso das instruções AES-NI, houve aumentos significativos nos valores de vazão alcançados pelas versões CBC(AESNI) e CTR(AESNI). Por isso, comparativamente, a versão EncFS++ apresenta perdas nos níveis de vazão nas operações de leitura aleatória e ganhos mais modestos nas demais operações. Na comparação com a versão CBC(AESNI), o ganho médio nas operações de leitura sequencial, escrita sequencial e escrita aleatória foi de 51,10%. Já na leitura aleatória houve perda de 43,46%. Na comparação com a versão CTR(AESNI), o ganho médio de vazão nas três operações foi de 13,60%. Na leitura aleatória também houve perda média de 44,07%.

A utilização da aceleração via hardware oferecida pelas instruções AES-NI se refletiram em uma menor utilização de CPU e consequentemente em uma maior eficiência de seu uso. Devido a este fato, neste caso houve um resultado inesperado, onde a utilização do processamento em GPU, na prática, não resultou em redução na utilização de CPU. Comparado à versão CBC(AESNI), considerando a média das quatro operações, a redução na eficiência de utilização da CPU foi de 25,72%. Comparado à versão CTR(AESNI), a redução média chegou a 45,40%.

A dificuldade para uso efetivo da GPU, em relação aos cenários que utilizam AES-NI, está relacionada a diversos fatores, sendo um deles a latência de disparo dos *kernels* CUDA. Além disso, com a utilização de uma ferramenta de análise de desempenho, observou-se que o uso de *spinlocks* na WAESlib tem uma sobrecarga considerável. Contudo, mesmo tendo sido utilizada uma primeira implementação da fila de prioridades na WAESlib com essas limitações iniciais, foi possível demonstrar que o uso da cifragem especulativa resultou em ganho de desempenho em

diversos cenários. Espera-se que a análise dos pontos críticos na WAESlib, apontados por essa ferramenta, possa contribuir para a solução dessas sobrecargas, principalmente nesse contexto de SACs.

Nas comparações de desempenho do EncFS++ com a versão que utilizou o cifrador mínimo, foi possível perceber que o EncFS++ conseguiu atingir níveis de vazão próximos dos valores máximos possíveis de serem alcançados na prática e considerando a atual arquitetura do SAC. Merecem destaque os resultados obtidos nas requisições de 4 KiB, onde foi possível chegar a 86%, 94% e 95% do limite máximo de vazão nas operações de leitura sequencial, escrita sequencial e escrita aleatória, respectivamente. Esses resultados podem ser vistos no apêndice D.

Os bons resultados obtidos nas operações de leitura sequencial, escrita sequencial e escrita aleatória comprovam a eficiência das técnicas de utilização dos *pools* de contextos para escrita e leitura, bem como da janela deslizante de contextos. Mesmo com o SAB armazenado em memória, seu mecanismo de funcionamento conseguiu garantir a produção das máscaras com a antecedência adequada. Neste sentido, a utilização de prioridades e a agregação de contextos oferecida pela WAESlib se mostraram essenciais. Também serviu para demonstrar que a latência de E/S não é a única forma de esconder a latência envolvida no processamento em GPU.

Os resultados negativos no caso da leitura aleatória serviram para demonstrar a importância de como os *pools* de contextos devem ser projetados e gerenciados. Se a definição dos contextos contidos no *pool* não for feita de forma apropriada, as máscaras não ficam prontas em tempo e o desempenho é comprometido. Também demonstra a importância de se evitar o disparo na produção de máscaras que não sejam efetivamente utilizadas.

Entretanto, as perdas foram mais significativas somente no cenário com o armazenamento em memória. Com o SAB armazenado em disco, a latência do acesso a disco contribui no sentido de dar mais tempo para as máscaras serem computadas na GPU e transferidas para a memória de sistema. Tanto que neste caso as perdas foram significativamente menores. De qualquer forma, o *pool* de contextos, no caso específico da leitura aleatória, precisa ser aperfeiçoado. Ele poderia, por exemplo, ter seu tamanho ajustado de forma dinâmica de acordo com o tamanho médio das requisições.

Capítulo 7

Conclusão

Os diversos modos de operação existentes podem ser empregados em várias aplicações nas funções de cifragem e decifragem simétricas de dados. O modo de operação CTR possui particularidades interessantes que quando bem exploradas permitem melhorar o desempenho dessas funções. Entre essas vantagens estão a capacidade de ser totalmente paralelizável e permitir a geração antecipada de máscaras de cifragem. Este trabalho procurou explorar ambas as vantagens, inclusive com a geração das máscaras de cifragem através do processamento em GPU.

A primeira parte se concentrou na implementação do modo CTR no SAC EncFS, já que por padrão ele utiliza somente o modo CBC. Nesta etapa foram atacadas questões importantes relacionadas a um componente indispensável ao modo CTR, denominado *nonce*. Essas questões tratam da sua geração, armazenamento e gerenciamento. O objetivo principal desta etapa foi garantir que essas mudanças não causassem quedas significativas no desempenho do SAC, as quais poderiam anular os benefícios posteriores do processamento em GPU. Também houve a preocupação em manter a segurança do modo de operação, utilizando uma forma determinística para a geração dos *nonces*, respeitando seu requisito de unicidade.

A segunda parte se concentrou em dotar o SAC da capacidade de executar suas funções criptográficas em GPU e, principalmente, aplicar a técnica de cifragem especulativa possível de ser realizada no modo CTR. Foram abordadas questões relacionadas com o gerenciamento dos contextos de cifragem, procurando criar mecanismos que garantissem a produção das máscaras de cifragem nos tempos adequados. Foram criados dois mecanismos diferentes, denominados *pools* de contextos: um para as operações de leitura e outro para as operações de escrita. O principal objetivo desta etapa foi justamente procurar explorar de forma eficiente a geração antecipada das máscaras de cifragem oferecida pela biblioteca WAESlib.

7.1 Resultados obtidos

Como forma de validar as técnicas apresentadas, elas foram implementadas no SAC originalmente denominado EncFS. Com relação a utilização do modo CTR, foram feitas análises de desempenho com medições envolvendo vazão em cenários de micro e *macrobenchmark*. Nestas análises foi comparado o desempenho do EncFS operando no seu modo padrão, o CBC, e o modo CTR implementado.

Os resultados apresentaram ganhos significativos de vazão principalmente nas operações de escrita e reescrita de dados tanto sequenciais quanto aleatórias. Houve ganhos modestos na operação de leitura sequencial e praticamente nulos no caso da leitura aleatória. Estes resultados comprovam os benefícios provenientes da total paralelização oferecida pelo modo

CTR, principalmente nos processos de cifragem utilizados nas operações de escrita. Já os resultados modestos na leitura sequencial e nulos na aleatória são explicados pela capacidade de paralelização que também é possível no modo CBC, o que diminui as vantagens da simples utilização do modo CTR.

Também foram feitos testes de *macrobenchmark* medindo tempo de cópia de arquivos com variados tamanhos e quantidades. Houve uma melhora geral nos tempos envolvidos na maioria dos cenários. A única exceção foram arquivos muitos pequenos, por volta de 512 bytes. Nos testes de latência também houve resultados positivos em praticamente todos os cenários medidos, com a única exceção também sendo a escrita sequencial de arquivos muito pequenos. Em ambos os casos onde houve queda de desempenho, a explicação está associada à sobrecarga no gerenciamento dos *nonces* que se torna mais significativa nesses cenários.

Nas análises com processamento em GPU, foram feitos apenas testes de *microbenchmark* medindo vazão e utilização de CPU. Os testes compararam versões do EncFS operando no modo CBC padrão e no modo CTR implementado, ambos com processamento exclusivo em CPU e utilizando ou não as instruções AES-NI. Também foi comparada a versão do EncFS utilizando o modo CTR implementado e a biblioteca WAESlib, consequentemente realizando processamento heterogêneo em CPU e GPU. Essa versão foi denominada EncFS++.

Em ambientes sem AES-NI, os resultados apresentaram ganhos significativos de vazão nas operações de leitura e escrita sequenciais, bem como na escrita aleatória, com ganhos menores na leitura aleatória. Também houve melhora significativa na eficiência de uso da CPU, o que na prática significa que o EncFS++ conseguiu processar uma quantidade maior de dados por percentual de uso da CPU. O destaque desse cenário foi o aumento significativo na capacidade de vazão para as operações de leitura, algo não alcançado nas análises da primeira etapa onde somente foi utilizado o modo CTR, sem o processamento em GPU. Também demonstra a efetividade na exploração da característica de cômputo antecipado de máscaras oferecida pelo modo CTR.

Os resultados comprovam que em um ambiente com CPUs sem suporte a aceleração de funções criptográficas, a utilização da biblioteca WAESlib para processamento em GPU se mostra bastante eficiente. Inclusive para acelerar operações com pequenas quantidades de dados, como no caso de requisições de apenas 4 KiB, algo apontado como não viável em pesquisas anteriores. Os resultados também demonstram a potencialidade da biblioteca no sentido de oferecer outras opções de cifradores simétricos de bloco. Isto seria extremamente útil para acelerar seu processamento, principalmente considerando que eles não teriam a opção de aceleração via instruções específicas de CPU, como ocorre com o AES.

Em ambientes com processadores que façam uso das instruções AES-NI, os ganhos de vazão foram modestos. Apesar disto e, considerando toda a sobrecarga envolvida no processamento em GPU, estes ganhos também merecem destaque. Neste cenário, o objetivo de se reduzir a utilização de CPU não foi alcançado, pelo contrário, em alguns cenários houve aumento significativo. Neste caso, é importante ressaltar que a questão de uso eficiente de CPU pela WAESlib ainda não foi atacada, tendo sido priorizado o desenvolvimento e aperfeiçoamento de outros mecanismos internos de funcionamento da biblioteca, bem como sua estabilidade, a fim de ser utilizada neste trabalho.

Já os bons resultados alcançados em relação à vazão comprovam a eficiência das técnicas de utilização dos *pools* de contextos, as quais foram utilizadas nos processos de leitura e escrita sequenciais e escrita aleatória. São técnicas que demonstram como os contextos podem ser gerenciados no sentido de se produzir as máscaras de cifragem com a antecedência adequada. As técnicas se mostraram eficientes até mesmo em ambientes de baixíssima latência, como foi o caso em que o SAB foi armazenado em memória.

As análises de desempenho que compararam o EncFS++ ao EncFS utilizando os cifradores mínimo e nulo ajudaram a demonstrar que a cifragem especulativa em GPU permitiu atingir níveis de vazão próximos ao limite prático possível, notadamente em requisições de 4 KiB e nas operações de leitura sequencial, escrita sequencial e escrita aleatória. Esses resultados ajudaram a demonstrar que aumentos de desempenho mais significativos somente seriam possíveis com o aperfeiçoamento na arquitetura do próprio SAC como, por exemplo, alterando-o para que passasse a operar no espaço do *kernel*.

A exceção nos bons resultados ocorreu na operação de leitura aleatória, cuja simples utilização do *pool* de contextos para leitura, sem adaptações, apresentou queda nos níveis de vazão. Neste caso, pelo menos levando em consideração o padrão de acesso aleatório gerado pela ferramenta utilizada, o *pool* precisa ser aperfeiçoado. Este aperfeiçoamento poderia ser no sentido de torná-lo dinâmico, ajustando seu tamanho conforme algumas variáveis como, por exemplo, o tamanho médio das requisições.

7.2 Trabalhos futuros

Este trabalho pode ser aprimorado e estendido em, pelo menos, três áreas: segurança da informação, sistemas operacionais e análise de desempenho com processamento paralelo em GPU.

Em segurança da informação, seria interessante realizar análises de vulnerabilidades na forma como o modo CTR foi implementado neste contexto de SACs, tanto no sentido de detectá-las, quanto indicar formas de solucioná-las. Uma vulnerabilidade clara na solução atual implementada é a não aplicação de técnicas que procurem detectar alterações não autorizadas no contador e *nonces* armazenados. Também seria interessante estudar a utilização de outros modos de operação que ofereçam recursos adicionais de segurança, como autenticidade e integridade, e que sejam fortemente baseados no modo CTR, como o CGM.

Na área de sistemas operacionais, seria interessante criar um novo SAC, o qual seja projetado para trabalhar com o modo CTR. As técnicas criadas e desenvolvidas neste trabalho foram validadas através da adaptação do EncFS, o que limitou a flexibilidade das alterações. Com um projeto partindo do zero, a questão de geração e armazenamento de *nonces* poderia ser melhor integrada ao SAC. Além disso, como no espaço do usuário o caminho mais fácil para a criação de um sistema de arquivos é a utilização da biblioteca libfuse, poder-se-ia analisar os possíveis benefícios de se utilizar os recursos da API de baixo nível da biblioteca, ao contrário do EncFS que utilizada a API de alto nível.

Apesar do processamento em GPU já ter sido estudado em SACs no espaço do *kernel*, nenhum deles utilizou o modo CTR e a biblioteca WAESlib. Considerando que o processamento em GPU permite atingir altos níveis de vazão, seria interessante analisar, por exemplo, os ganhos provenientes de uma possível integração do dm-crypt com a WAESlib, ou até mesmo da Crypto-API do *kernel* Linux.

Por fim, na área de análise de desempenho, seria interessante submeter o SAC a cargas variadas de trabalho, seja através de *macrobenchmarks* já existentes ou criados, os quais procurem simular cargas reais de trabalho. Isto seria importante principalmente para analisar melhor o comportamento e eficácia dos *pools* de contextos criados e utilizados com a WAESlib, principalmente no caso da leitura aleatória. Com certeza há margem para aprimoramentos, incluindo até mesmo a criação de novos *pools* com mecanismos de funcionamento diferentes dos propostos neste trabalho.

Referências Bibliográficas

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

[2] 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.

[3] Herb Sutter and James Larus. Software and the Concurrency Revolution. *Queue*, 3(7):54–62, September 2005.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Vandeir Eduardo e Luis Carlos E. de Bona e Wagner M. Nunan Zola. Utilização de Criptografia em Modo CTR Aplicada ao Armazenamento de Arquivos em Nuvem. *Workshop em Clouds e Aplicações (WCGA - SBRC)*, 16, 2018.

[16] William Stallings. *Criptografia e segurança de redes: princípios e práticas*. Person Education do Brasil, São Paulo, SP, 6th edition, 2015.

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

[18] C. E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, Oct 1949.

[19] National Institute of Standards and Technology. Federal Information Processing Standards Publication (FIPS PUB) 197: Advanced Encryption Standard (AES). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf, 2001. Acessado em 01/05/2017.

[20] National Institute of Standards and Technology. Special Publication (SP) 800-38A: Recommendation for Block Cipher Modes of Operation (Methods and Techniques). http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf, 2001. Acessado em 02/05/2017.

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

[22] National Institute of Standards and Technology. Addendum to Special Publication (SP) 800-38A: Three Variants of Ciphertext Stealing for CBC Mode. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a-add.pdf, 2010. Acessado em 02/05/2017.

[23] Marco Macchetti, Mario Caironi, Luca Breveglieri, and Alessandra Cherubini. A Complete Formulation of Generalized Affine Equivalence. In Mario Coppo, Elena Lodi, and G. Michele Pinna, editors, *Theoretical Computer Science: 9th Italian Conference, ICTCS 2005*, pages 338–347, Siena, Italy, 2005. Springer.

[24] Philipp Jovanovic. *Analysis and Design of Symmetric Cryptographic Algorithms*. PhD thesis, Faculdade de Ciência da Computação e Matemática, Universidade de Passau, Passau - Alemanha, 2015.

[25] Eli Biham, Ross Anderson, and Lars Knudsen. Serpent: A New Block Cipher Proposal. In Serge Vaudenay, editor, *Fast Software Encryption*, pages 222–238, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[26] Federal Agency on Technical Regulation and Metrology. Cryptographic Data Security: Block Ciphers (GOST R 34.12–2015). http://tc26.ru/en/standard/gost/GOST_R_34_12_2015_ENG.pdf, 2015. Acessado em 08/06/2017.

[27] Bruce Schneier. The Twofish: A 128-Bit Block Cipher. https://www.schneier.com/academic/archives/1998/06/twofish_a_128-bit_bl.html, 1998. Acessado em 08/06/2017.

[28] Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakajima, and Toshio Tokita. Specification of Camellia – a 128 bit Block Cipher. http://info.isl.ntt.co.jp/crypt/eng/camellia/dl/01espec.pdf, 2000. Acessado em 08/06/2017.

[29] Xuejia Lai and James L. Massey. A Proposal for a New Block Encryption Standard. In Ivan Bjerre Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, pages 389–404, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[30] P. Juno and S. Vaudena. FOX Specifications. http://crypto.junod.info/fox_spec_v1.2.pdf, 2005. Acessado em 08/06/2017.

[31] National Institute of Standards and Technology. Special Publication (SP) 800-38B: Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38b.pdf, 2005. Acessado em 08/06/2017.

[32] National Institute of Standards and Technology. Special Publication (SP) 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf, 2007. Acessado em 02/05/2017.

[33] National Institute of Standards and Technology. Special Publication (SP) 800-38E: Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38e.pdf, 2010. Acessado em 02/05/2017.

[34] A. S. Tanenbaum. *Modern Operating Systems*. Pearson, 4th edition, 2014.

[35] Robert Love. *Linux Kernel Development*. Addison-Wesley, 3rd edition, 2010.

[36] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel: From I/O Ports to Process Managemen*. O'Reilly, 3rd edition, 2005.

[37] Wikipedia Contributors. Comparison of disk encryption software. https://en.wikipedia.org/w/index.php?title=Comparison_of_disk_encryption_software&oldid=769085202, 2017. Acessado em 07/03/2017.

[38] OpenSSL Software Foundation. OpenSSL Cryptography and SSL/TLS Toolkit. https://www.openssl.org/, 2017. Acessado em 07/03/2017.

[39] The GNUPG Project. The GNU Privacy Guard. https://www.gnupg.org/, 2017. Acessado em 07/03/2017.

[40] The Document Foundation. LibreOffice. https://www.libreoffice.org/, 2017. Acessado em 07/03/2017.

[41] N. Rath and M. Szeredi. The reference implementation of the Linux FUSE interface (libfuse). https://github.com/libfuse/libfuse, 2017. Acessado em 07/03/2017.

[42] Valient Gough. EncFS: an Encrypted Filesystem for FUSE. https://github.com/vgough/encfs, 2017. Acessado em 07/03/2017.

[43] Sebastian Messmer. CryFS: A Criptographic Filesystem for the cloud. https://www.cryfs.org/, 2017. Acessado em 08/03/2017.

[44] Sebastian Messmer. Design and Implementation of a Provably Secure Encrypted Cloud Filesystem. Master's thesis, Departamento de Informática, Instituto Tecnológico de Karlshule, Karlshule, Alemanha, 2015.

[45] Matt Blaze. A cryptographic file system for unix. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 9–16, New York, NY, USA, 1993. ACM.

[46] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano. The Design and Implementation of a Transparent Cryptographic File System for UNIX. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 199–212, Berkeley, CA, USA, 2001. USENIX Association.

[47] C. P. Wright, M. C. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, USA, 2003. USENIX Association.

[48] E. Zadok and J. Nieh. FiST: a language for stackable file systems. In *Proceedings of the Annual USENIX Technical Conference (ATEC '00)*, page 5, Berkeley, USA, 2000. USENIX Association.

[49] Erez Zadok, Ion Badulescu, and Alex Shender. Cryptfs: A Stackable Vnode Level Encryption File System. Technical report, Department of Computer Science, Columbia University, New York, USA, 1998.

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

[51] Milan Broz. dm-crypt: Linux kernel device-mapper crypto target. https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt, 2017. Acessado em 09/03/2017.

[52] Milan Broz. Linux Unified Key Setup. https://gitlab.com/cryptsetup/cryptsetup/, 2017. Acessado em 09/03/2017.

[53] Clemens Fruhwirth. LUKS On-Disk Format Specification. https://gitlab.com/cryptsetup/cryptsetup/wikis/LUKS-standard/on-disk-format.pdf, 2017. Acessado em 08/03/2017.

[54] Idrix. VeraCrypt Home Page. https://veracrypt.codeplex.com/, 2017. Acessado em 10/03/2017.

[55] Mounir Idrassi. VeraCrypt Introduction. https://veracrypt.codeplex.com/wikipage?title=Introduction, 2017. Acessado em 10/03/2017.

[56] Mounir Idrassi. VeraCrypt Volume Format Specification. https://veracrypt.codeplex.com/wikipage?title=VeraCrypt%20Volume%20Format%20Specification, 2017. Acessado em 10/03/2017.

[57] Mounir Idrassi. VeraCrypt Encryption Scheme. https://veracrypt.codeplex.com/wikipage?title=Encryption%20Scheme, 2017. Acessado em 10/03/2017.

[58] Microsoft. The Encrypting File System. Disponível. https://technet.microsoft.com/en-us/library/cc700811.aspx, 2017. Acessado em 09/03/2017.

[59] Microsoft. BitLocker Driver Encryption Overview. https://technet.microsoft.com/en-us/library/cc732774(v=ws.11).aspx, 2017. Acessado em 09/03/2017.

[60] David Mazières. *Self-certifying File System*. PhD thesis, Departamento de Engenharia Elétrica e Ciência da Computação, Instituto Tecnológico de Massachusets, Cambridge, EUA, 2000.

[61] Kevin E. Fu. Group Sharing and Random Access in Cryptographic Storage File System. Master's thesis, Departamento de Informática, Instituto Tecnológico de Karlshule, Cambridge, EUA, 1998.

[62] E. Miller, D. Long, W. Freeman, and B. Reed. Strong security for distributed file systems. In *Conference Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference (IPCCC 2001)*, pages 34–40, Phoenix, USA, Apr 2001.

[63] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 29–42, Berkeley, CA, USA, 2003. USENIX Association.

[64] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*, pages 131–145. Internet Society (ISOC), February 2003.

[65] Vishal Kher and Yongdae Kim. Securing Distributed Storage: Challenges, Techniques, and Systems. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability*, StorageSS '05, pages 9–25, New York, NY, USA, 2005. ACM.

[66] Sarah M. Diesburg and An-I Andy Wang. A Survey of Confidential Data Storage and Deletion Methods. *ACM Comput. Surv.*, 43(1):2:1–2:37, December 2010.

[67] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010.

[68] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2013.

[69] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. John Wiley & Sons, 2014.

[70] NVidia Corporation. NVIDIA Tesla P100 White Paper. http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper-v1.2.pdf, 2017. Acessado em 23/05/2017.

[71] NVidia Corporation. CUDA C Programming Guide (Versão PG-02829-001_v8.0). http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf, 2017. Acessado em 23/05/2017.

[72] NVidia Corporation. NVIDIA Kepler GK110/210 White Paper. http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf, 2017. Acessado em 24/05/2017.

[73] Luke Durant, Olivier Giroux, Mark Harris, and Nick Stam. Inside Volta: The World's Most Advanced Data Center GPU. https://devblogs.nvidia.com/parallelforall/inside-volta/, 2017. Acessado em 24/05/2017.

[74] NVidia Corporation. NVIDIA Fermi White Paper. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, 2017. Acessado em 24/05/2017.

[75] Mark Harris. Maxwell: The Most Advanced CUDA GPU Ever Made. https://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/, 2017. Acessado em 24/05/2017.

[76] Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015.

[77] Advanced Micro Devices. AMD Graphics Core Next (GCN) Architecture (White paper). https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf, 2017. Acessado em 06/08/2017.

[78] Intel Corporation. The Compute Architecture of Intel Processor Graphics Gen9. https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf, 2017. Acessado em 06/08/2017.

[79] Intel Corporation. Intel Open Source HD Graphics and Intel Iris Plus Graphics Programmer's Reference Manual. https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-kbl-vol04-configurations.pdf, 2017. Acessado em 06/08/2017.

[80] Ian Cutress. AMD Announces the 7th Generation APU: Excavator mk2 in Bristol Ridge and Stoney Ridge for Notebooks. http://www.anandtech.com/show/10362/amd-7th-generation-apu-bristol-ridge-stoney-ridge-for-notebooks, 2017. Acessado em 06/08/2017.

[81] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[82] Khronos Group. OpenCL Overview. https://www.khronos.org/opencl/, 2017. Acessado em 06/08/2017.

[83] Erik Smistad, Thomas L. Falch, Mohammadmehdi Bozorgi, Anne C. Elster, and Frank Lindseth. Medical image segmentation on GPUs - A comprehensive review. *Medical Image Analysis*, 20(1):1–18, 2015.

[84] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC — First Experiences with Real-World Applications. In Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, pages 859–870, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[85] OpenACC Organization. About OpenACC. https://www.openacc.org/about, 2017. Acessado em 06/08/2017.

[86] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kolodziej, and Christoph Kessler. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, ARMS-CC '17, pages 1–6, New York, NY, USA, 2017. ACM.

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

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

[89] 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.

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

[91] Weibin Sun. *Harnessing GPU Computing in System-Level Software*. PhD thesis, Escola de Computação, Universidade de Utah, Utah - EUA, 2014.

[92] Leslie Xu. Securing the Enterprise with Intel AES-NI. https://www.intel.com/content/www/us/en/enterprise-security/enterprise-security-aes-ni-white-paper.html, 2010. Acessado em 20/07/2018.

Apêndice A

Medições de vazão

Este apêndice contém tabelas com valores de vazão obtidos em algumas das análises de desempenho discutidas na Seção 5.3.

A.1 Vazão com múltiplos processos e acesso a disco

A.1.1 Escrita sequencial

Tabela A.1: Níveis de vazão para escrita sequencial com 1 processo.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	1	4.148	4.316	60,61	4.229,2	5.699	6.154	149,4	5.898,2	39,46
	4	10.302	10.610	106,39	10.403,6	22.815	24.550	579,52	23.705,6	127,86
	8	13.910	14.845	357,46	14.296,4	32.250	35.289	1.145,11	33.858,8	136,83
	16	17.237	17.890	249,58	17.524,6	36.421	44.206	3.066,67	39.897	127,66
	32	19.227	20.821	585,79	19.873,4	46.824	56.099	3.422,02	51.027	156,76
	64	20.690	21.364	239,32	21.024,2	53.757	61.128	2.745,88	57.087,6	171,53
	128	21.217	22.698	501,5	21.817,2	58.042	65.450	2.701,85	62.816,6	187,92
	256	21.240	22.541	453,96	21.910	57.286	61.646	1.533,92	58.736,2	168,08
	512	21.455	22.456	362,59	21.870,6	58.956	66.628	2.894,75	64.314,4	194,07
	Média	**16.602,89**	**17.504,56**	**324,13**	**16.994,36**	**41.338,89**	**46.794,44**	**2.026,57**	**44.149,04**	**159,79**

Tabela A.2: Níveis de vazão para escrita sequencial com 2 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
2	1	18.441	18.596	57,35	18.512,2	21.218	21.291	26,25	21.262,4	14,86
	4	20.936	21.234	117,32	21.116,2	62.060	62.498	173,38	62.306,8	195,07
	8	21.098	21.365	97,98	21.243,2	70.514	71.444	315,55	70.878,2	233,65
	16	22.883	23.445	206,22	23.083,8	76.632	77.028	143,51	76.908,2	233,17
	32	24.305	24.561	96,56	24.425,8	81.300	82.305	347,23	81.750,4	234,69
	64	24.840	25.181	130,25	25.038,6	83.650	84.128	190,3	83.897,8	235,07
	128	25.259	25.688	155,16	25.521,6	85.517	85.886	140,46	85.723,6	235,89
	256	25.462	25.976	164,86	25.723	85.982	86.727	294,54	86.385,4	235,83
	512	25.412	25.878	157,61	25.627,2	86.163	86.516	135,59	86.313	236,8
	Média	**23.181,78**	**23.547,11**	**131,48**	**23.365,73**	**72.559,56**	**73.091,44**	**196,31**	**72.825,09**	**211,67**

Tabela A.3: Níveis de vazão para escrita sequencial com 4 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
4	1	21.752	21.966	70,37	21.838	24.337	24.755	174,15	24.526,8	12,31
	4	26.071	26.160	34,86	26.119,4	71.563	76.231	1.860,54	73.954,6	183,14
	8	26.231	26.313	29,67	26.274,6	84.168	104.410	7.397,5	91.884,8	249,71
	16	26.224	26.482	89,11	26.367,4	93.626	111.512	6.355,78	105.904	301,65
	32	23.943	26.190	888,26	25.718,8	103.360	117.216	5.467,34	108.334	321,22
	64	25.814	26.165	113,19	26.010	104.757	119.324	5.627,24	113.818	337,59
	128	26.020	26.163	46,95	26.084,2	91.288	105.021	4.840,58	99.947,6	283,17
	256	26.027	26.191	70,68	26.114	101.413	120.876	6.815,86	110.648,2	323,71
	512	26.038	26.209	59,27	26.147,6	112.915	127.526	5.034,1	120.076,4	359,23
	Média	**25.346,67**	**25.759,89**	**155,82**	**25.630,44**	**87.491,89**	**100.763,44**	**4.841,46**	**94.343,82**	**268,09**

Tabela A.4: Níveis de vazão para escrita sequencial com 8 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
8	1	22.343	22.568	79,91	22.439,2	25.652	26.011	115,62	25.804,8	15
	4	26.114	26.271	57	26.198,4	76.004	91.596	5.961,48	82.847,8	216,23
	8	26.250	26.534	93,98	26.423,2	89.525	96.957	3.182,7	92.893,4	251,56
	16	26.425	26.577	53,08	26.505,8	90.737	101.288	3.613,25	94.605,6	256,92
	32	26.566	26.743	60,52	26.663,2	93.896	105.059	3.743,59	99.357,8	272,64
	64	26.543	26.675	44,78	26.592,6	98.901	111.538	4.301,88	105.112	295,27
	128	26.396	26.780	142,03	26.598	99.090	105.613	2.390,31	102.462,2	285,23
	256	26.552	26.783	78,26	26.668,6	97.404	111.962	4.946	104.246,2	290,89
	512	26.575	26.775	67,02	26.656,4	96.545	110.241	4.787,45	105.662,2	296,39
	Média	**25.973,78**	**26.189,56**	**75,17**	**26.082,82**	**85.306**	**95.585**	**3.671,36**	**90.332,44**	**246,33**

A.1.2 Escrita aleatória

Tabela A.5: Níveis de vazão para escrita aleatória com 1 processo.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	1	146	151	1,96	149,6	135	151	6,09	143,6	-4,01
	4	525	549	8,7	536,2	3.617	3.670	19,93	3.653,6	581,39
	8	1.050	1.060	4,12	1.056,8	5.233	5.325	32,85	5.288,2	400,4
	16	1.870	1.880	3,44	1.875,4	8.354	8.528	60,96	8.422,8	349,12
	32	3.422	3.446	7,76	3.432,4	13.594	14.244	232,27	13.948	306,36
	64	5.624	5.732	42,7	5.680,6	20.092	20.434	144,11	20.263	256,71
	128	8.665	8.713	15,77	8.685,6	27.493	28.137	223,17	27.725,4	219,21
	256	12.171	12.383	89,96	12.297,2	33.628	36.291	915,01	34.675,8	181,98
	512	15.231	15.656	155,77	15.537,6	40.589	44.890	1.545,41	41.966,4	170,1
	Média	**5.411,56**	**5.507,78**	**36,69**	**5.472,38**	**16.970,56**	**17.963,33**	**353,31**	**17.342,98**	**216,92**

Tabela A.6: Níveis de vazão para escrita aleatória com 2 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
2	1	136	138	0,98	136,8	126	138	4,63	133,4	-2,49
	4	516	518	0,75	516,8	4.047	4.072	8,38	4.059,2	685,45
	8	977	1.043	25,08	1.010,8	5.660	5.732	27,2	5.708,4	464,74
	16	1.810	1.846	13,15	1.820,6	8.862	9.060	65,62	8.976,6	393,06
	32	3.260	3.310	16,49	3.283,2	14.457	14.596	48,9	14.543,6	342,97
	64	5.622	5.715	30,35	5.666,2	23.317	23.506	68,57	23.385	312,71
	128	8.782	9.074	123,77	8.944,6	33.273	33.613	127,51	33.448	273,95
	256	12.753	12.948	69,51	12.863,8	42.806	43.539	281,72	43.208,2	235,89
	512	16.867	16.967	35,68	16.918	53.967	54.617	212,83	54.282,8	220,86
	Média	**5.635,89**	**5.728,78**	**35,08**	**5.684,53**	**20.723,89**	**20.985,89**	**93,93**	**20.860,58**	266,97

Tabela A.7: Níveis de vazão para escrita aleatória com 4 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
4	1	120	126	2,4	122,2	121	127	2,23	124,8	2,13
	4	460	476	6,01	467,2	3.996	4.076	27,41	4.033,6	763,36
	8	921	953	13,18	934,6	5.631	5.686	24,02	5.660,2	505,63
	16	1.848	1.877	10,65	1.865,2	8.819	8.949	51,21	8.872,8	375,7
	32	3.174	3.249	25,58	3.213,2	14.316	14.579	85,63	14.459,8	350,01
	64	5.431	5.462	10,55	5.444,8	23.200	23.553	134,68	23.376,4	329,33
	128	8.092	8.416	118,57	8.264,8	33.389	34.391	345,74	33.951,8	310,8
	256	12.074	12.240	62,73	12.115,6	47.923	49.458	530,02	48.653	301,57
	512	16.955	17.131	64,92	17.024	60.860	66.174	2.499,52	63.082	270,55
	Média	**5.452,78**	**5.547,78**	**34,95**	**5.494,62**	**22.028,33**	**22.999,22**	**411,16**	**22.468,27**	308,91

Tabela A.8: Níveis de vazão para escrita aleatória com 8 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
8	1	120	125	1,94	121,2	124	130	2,14	126,2	4,13
	4	441	483	18,24	463,6	3.881	3.935	18,69	3.909,8	743,36
	8	848	905	18,58	877,4	5.437	5.492	18,1	5.469,6	523,39
	16	1.676	1.818	52,77	1.717,6	8.693	9.261	215,59	8.913,6	418,96
	32	3.268	3.296	10,8	3.283,4	14.197	14.396	71,01	14.321	336,16
	64	5.522	5.646	40,42	5.584	22.138	22.723	212,83	22.499,8	302,93
	128	8.259	8.482	90,48	8.376,8	33.879	34.520	242,33	34.152	307,7
	256	11.308	11.946	261,15	11.620,8	47.686	48.974	518,82	48.219,2	314,94
	512	15.290	16.028	238,24	15.642,8	63.136	66.824	1.312,02	65.616,4	319,47
	Média	**5.192,44**	**5.414,33**	**81,4**	**5.298,62**	**22.130,11**	**22.917,22**	**290,17**	**22.580,84**	326,16

A.1.3 Leitura sequencial

Tabela A.9: Níveis de vazão para leitura sequencial com 1 processo.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	1	129.980	131.387	497,23	130.540,8	149.728	153.948	1.534,16	152.247,4	16,63
	4	127.279	128.250	336,04	127.611,6	148.305	150.415	773,36	149.786,4	17,38
	8	128.300	129.364	409,81	128.952,8	146.612	149.489	1.029,05	148.050,4	14,81
	16	127.800	129.568	638,87	128.666	146.777	149.967	1.122,31	147.851,2	14,91
	32	126.176	128.527	897,79	127.577,4	144.384	147.040	882,08	145.763,4	14,25
	64	124.403	126.835	818,63	125.730,8	145.151	146.580	516,67	145.753,6	15,93
	128	124.664	126.030	571,29	125.386,6	144.035	145.732	709,71	144.891,6	15,56
	256	124.640	128.602	1.408,64	126.065,2	143.499	146.285	939,43	144.919,6	14,96
	512	124.830	128.628	1.359,79	126.011	142.593	146.580	1.497,97	144.724,4	14,85
	Média	**126.452,44**	**128.576,78**	**770,9**	**127.393,58**	**145.676**	**148.448,44**	**1.000,53**	**147.109,78**	15,48

Tabela A.10: Níveis de vazão para leitura sequencial com 2 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
2	1	158.798	161.737	1.219,67	160.228	161.319	164.004	1.054,76	162.578,4	1,47
	4	159.203	161.897	946,5	160.530,4	160.923	162.781	651,97	161.911,4	0,86
	8	152.533	157.956	1.941,19	156.278	157.292	161.837	1.492,76	159.810,6	2,26
	16	151.231	157.179	1.941,8	153.777,8	158.932	160.785	712,32	159.554,6	3,76
	32	152.693	157.747	2.156,84	155.015,8	156.934	161.617	1.785,73	158.983	2,56
	64	153.444	157.217	1.298,74	155.290,6	156.317	160.943	1.663,71	158.661,4	2,17
	128	154.693	157.785	1.114,87	156.829,8	157.519	160.745	1.135,9	158.948,2	1,35
	256	150.380	156.503	2.185,83	154.575	158.548	163.309	1.692,03	160.342	3,73
	512	155.023	157.122	761,92	156.123	156.000	160.588	1.711,43	159.132,6	1,93
	Média	**154.222**	**158.349,22**	**1.507,48**	**156.516,49**	**158.198,22**	**161.845,44**	**1.322,29**	**159.991,36**	2,22

Tabela A.11: Níveis de vazão para leitura sequencial com 4 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
4	1	127.128	129.224	706,77	128.295,6	133.576	135.629	783,94	134.461,8	4,81
	4	129.672	132.132	926,77	130.638,8	131.578	135.608	1.463,9	134.229,4	2,75
	8	123.921	132.395	3.197,71	126.975,6	132.169	135.307	1.184,46	133.699,6	5,3
	16	123.344	129.109	1.851,13	126.340,4	132.570	137.485	1.664,9	134.808,2	6,7
	32	127.087	132.302	1.801,25	130.362,8	132.102	134.577	920,58	133.337,8	2,28
	64	117.243	129.160	4.024,4	124.632	131.314	134.391	1.284,36	132.930	6,66
	128	123.910	127.477	1.280,85	125.186,6	131.486	135.952	1.515,71	133.049,4	6,28
	256	124.085	136.647	4.340,1	128.687,4	131.545	136.832	1.876,52	133.458	3,71
	512	123.332	133.155	3.255,39	127.476,6	132.449	136.171	1.266,74	134.110	5,2
	Média	**124.413,56**	**131.289**	**2.376,04**	**127.621,76**	**132.087,67**	**135.772,44**	**1.329,01**	**133.787,13**	4,83

Tabela A.12: Níveis de vazão para leitura sequencial com 8 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
8	1	106.478	109.393	1.105,41	107.811,4	112.718	115.441	1.058,51	113.863,4	5,61
	4	107.867	110.662	1.076,81	108.779	111.662	115.512	1.327,81	113.934,4	4,74
	8	100.972	110.058	3.364,53	106.611	113.401	115.816	822,51	114.782,6	7,66
	16	104.221	108.728	1.734,25	106.290,4	110.990	118.170	2.614,54	113.519,2	6,8
	32	102.037	108.276	2.313,81	106.386,2	112.837	114.927	765,77	114.054,6	7,21
	64	105.143	108.798	1.375,3	106.670	111.465	115.489	1.439,69	114.262,4	7,12
	128	100.544	108.870	2.989,32	104.622,6	111.809	115.846	1.471,08	114.284,2	9,23
	256	105.046	108.927	1.497,05	106.832,8	112.631	114.849	950,43	113.551,4	6,29
	512	103.778	109.019	1.869,26	106.470,4	112.469	116.026	1.226,59	113.747,4	6,83
	Média	**104.009,56**	**109.192,33**	**1.925,08**	**106.719,31**	**112.220,22**	**115.786,22**	**1.297,44**	**113.999,96**	6,82

A.1.4 Leitura aleatória

Tabela A.13: Níveis de vazão para leitura aleatória com 1 processo.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	1	170	170	0	170	171	171	0	171	0,59
	4	666	666	0	666	668	669	0,4	668,8	0,42
	8	1.303	1.307	1,26	1.305	1.300	1.304	1,36	1.301,6	-0,26
	16	3.113	3.130	5,95	3.121,2	3.118	3.125	2,61	3.122	0,03
	32	11.140	11.171	12,52	11.158,4	11.303	11.400	40,24	11.344,8	1,67
	64	17.930	18.027	34,16	17.983	18.280	18.418	52,82	18.344,6	2,01
	128	24.187	24.345	57,87	24.254,2	24.705	25.192	165,44	24.972,8	2,96
	256	33.639	33.804	62,12	33.708,8	36.297	36.512	77,37	36.422,8	8,05
	512	50.031	50.763	237,1	50.387,2	57.858	58.104	97,99	57.970,6	15,05
	Média	**15.797,67**	**15.931,44**	**45,66**	**15.861,53**	**17.077,78**	**17.210,56**	**48,69**	**17.146,56**	8,1

Tabela A.14: Níveis de vazão para leitura aleatória com 2 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
2	1	159	160	0,4	159,8	154	157	0,98	155,8	-2,5
	4	628	637	3,6	629,8	620	622	0,63	621	-1,4
	8	1.171	1.181	3,54	1.177,2	1.151	1.156	1,85	1.154,6	-1,92
	16	2.407	2.436	10,86	2.422,6	2.380	2.391	4,34	2.386	-1,51
	32	5.070	5.142	29,11	5.096,8	5.018	5.071	20,71	5.040,4	-1,11
	64	12.810	12.952	60,4	12.875,2	12.525	12.687	60,56	12.624,6	-1,95
	128	23.312	23.539	82,83	23.456	23.142	23.271	45,69	23.192,4	-1,12
	256	32.118	32.345	81,82	32.219,4	31.950	32.138	73,99	32.045	-0,54
	512	51.220	51.702	178,07	51.357,6	50.876	51.012	46,74	50.950	-0,79
	Média	**14.321,67**	**14.454,89**	**50,07**	**14.377,16**	**14.201,78**	**14.278,33**	**28,39**	**14.241,09**	-0,95

Tabela A.15: Níveis de vazão para leitura aleatória com 4 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
4	1	144	152	2,93	147,8	147	151	1,47	148,8	0,68
	4	577	607	10,46	589,8	586	606	7,49	596,2	1,09
	8	1.096	1.105	3,83	1.099,4	1.090	1.099	3,82	1.094,8	-0,42
	16	2.191	2.208	5,59	2.199	2.176	2.192	5,71	2.186,6	-0,56
	32	4.155	4.176	8,98	4.163,2	4.093	4.154	20,96	4.131,2	-0,77
	64	8.384	8.515	46,07	8.468,8	8.352	8.456	33,75	8.401,8	-0,79
	128	14.974	15.132	56,94	15.050,2	14.605	14.922	107,51	14.775	-1,83
	256	26.292	26.647	118,84	26.483,6	26.124	26.505	152,43	26.323,2	-0,61
	512	46.983	47.763	262,95	47.305,4	47.065	47.714	214,53	47.373,2	0,14
	Média	**11.644**	**11.811,67**	**57,4**	**11.723,02**	**11.582**	**11.755,44**	**60,85**	**11.670,09**	-0,45

Tabela A.16: Níveis de vazão para leitura aleatória com 8 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
8	1	151	153	0,63	152	150	154	1,36	152,6	0,39
	4	607	610	1,1	608	608	612	1,83	610,2	0,36
	8	1.077	1.091	4,76	1.084,6	1.085	1.097	4,36	1.089,6	0,46
	16	2.120	2.151	11,75	2.128,2	2.118	2.142	8,29	2.132	0,18
	32	3.973	3.994	7,19	3.980,8	3.954	3.991	12,92	3.975,8	-0,13
	64	7.545	7.643	31,91	7.593	7.532	7.583	18,67	7.559,8	-0,44
	128	13.027	13.115	34,21	13.088,2	12.929	13.028	39,97	12.972,8	-0,88
	256	23.853	24.084	87,86	23.990,8	23.683	24.080	141,8	23.921,2	-0,29
	512	41.382	42.185	260,33	41.814,6	41.752	41.966	68,21	41.850,6	0,09
	Média	**10.415**	**10.558,44**	**48,86**	**10.493,36**	**10.423,44**	**10.517**	**33,05**	**10.473,84**	-0,19

A.2 Vazão com múltiplos processos, acesso a memória e múltiplos contextos

A.2.1 Leitura sequencial

Tabela A.17: Níveis de vazão para leitura sequencial com 1 processo.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	1	142.109	143.824	639,71	142.585,6	169.343	170.963	580,82	170.416,6	19,52
	4	138.798	139.240	143,72	138.984,4	166.405	169.343	1.048,63	167.660,2	20,63
	8	137.825	138.994	414,05	138.486,4	166.405	169.197	902,42	167.644	21,05
	16	138.164	138.700	171,58	138.456,2	166.264	167.325	346,36	166.899,8	20,54
	32	138.602	139.735	414,87	139.103,4	164.663	167.468	972,85	166.326,2	19,57
	64	138.505	139.339	286,92	138.808,4	165.843	167.183	477,42	166.716,6	20,11
	128	138.505	138.896	146,42	138.690,4	166.546	168.328	701,11	167.186	20,55
	256	137.632	139.290	551,95	138.360,6	165.704	167.682	662,3	166.492	20,33
	512	138.213	138.700	182,17	138.407,6	166.475	168.041	611,49	167.071,6	20,71
	Média	**138.705,89**	**139.635,33**	**327,93**	**139.098,11**	**166.405,33**	**168.392,22**	**700,38**	**167.379,22**	20,33

Tabela A.18: Níveis de vazão para leitura sequencial com 2 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
2	1	181.918	182.297	134,68	182.111,4	424.410	437.392	4.639,74	433.489	138,04
	4	181.749	181.918	62,19	181.842	415.222	420.551	1.741,34	417.566,4	129,63
	8	182.848	183.360	198,25	183.061,2	406.424	415.661	3.376,79	413.025,8	125,62
	16	182.848	183.317	163,18	183.129,4	409.173	414.566	1.772,74	412.183,4	125,08
	32	182.721	183.061	138,66	182.907,8	407.688	412.176	1.583,42	410.632,2	124,5
	64	182.382	183.104	264,94	182.882,6	410.027	413.911	1.534,52	412.224,6	125,4
	128	182.213	182.933	271,3	182.746,6	408.536	412.608	1.494,37	410.160,6	124,44
	256	182.170	183.104	313,44	182.755,2	413.259	418.760	2.081,47	414.619,8	126,87
	512	182.678	182.976	104,19	182.848,2	412.392	415.661	1.310,03	414.176,8	126,51
	Média	**182.391,89**	**182.896,67**	**183,43**	**182.698,27**	**411.903,44**	**417.920,67**	**2.170,49**	**415.342,07**	127,34

Tabela A.19: Níveis de vazão para leitura sequencial com 4 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
4	1	181.876	182.403	181,1	182.179	637.819	641.200	1.078,83	639.376,8	250,96
	4	181.749	182.276	196,42	182.128,4	691.065	699.050	2.680,53	694.799,2	281,49
	8	182.002	182.466	156,94	182.250,6	690.155	709.456	7.303,75	700.496,2	284,36
	16	182.086	182.763	229,83	182.339,8	704.688	709.776	1.935,41	706.719,6	287,58
	32	182.255	182.509	105,09	182.411,4	702.171	706.587	1.652,57	704.186,8	286,04
	64	182.107	182.466	127,76	182.356,4	699.672	710.096	3.462,76	704.957,4	286,58
	128	182.044	182.509	162,89	182.208,8	701.858	711.059	3.134,9	707.491,2	288,29
	256	181.435	182.192	274,16	181.972,8	699.672	705.636	2.241,38	703.497,2	286,59
	512	181.876	182.065	66,94	181.955,8	693.197	699.361	2.147,6	696.518,4	282,8
	Média	**181.936,67**	**182.405,44**	**166,79**	**182.200,33**	**691.144,11**	**699.135,67**	**2.848,64**	**695.338,09**	281,63

Tabela A.20: Níveis de vazão para leitura sequencial com 8 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
8	1	178.612	179.961	462,66	179.277,2	696.573	701.545	1.829,35	699.117,2	289,96
	4	179.305	179.653	118,49	179.448	758.738	767.437	3.246,23	764.392,8	325,97
	8	179.714	179.951	75,72	179.837,6	772.336	779.610	2.729,89	776.080,6	331,55
	16	179.448	179.838	128,53	179.675,6	770.068	782.519	4.812,92	776.177,4	331,99
	32	178.815	180.095	446,95	179.532,8	773.666	782.714	3.407,42	777.928,4	333,31
	64	179.550	179.940	132,71	179.702	774.047	780.190	2.156,08	777.996,4	332,94
	128	179.029	179.663	225,39	179.407,2	767.437	778.645	3.743,43	773.341,8	331,05
	256	179.202	179.704	175,63	179.476,4	762.970	766.689	1.398,34	765.051	326,27
	512	178.805	179.550	265,68	179.304,8	752.566	762.231	3.267,19	758.239,6	322,88
	Média	**179.164,44**	**179.817,22**	**225,75**	**179.517,96**	**758.711,22**	**766.842,22**	**2.954,54**	**763.147,24**	325,1

A.2.2 Escrita sequencial

Tabela A.21: Níveis de vazão para escrita sequencial com 1 processo.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
1	1	4.609	4.911	108,83	4.766	6.354	7.049	242,37	6.582,6	38,12
	4	36.742	40.998	1.849,5	39.098	47.204	49.535	901,85	48.268,2	23,45
	8	40.193	45.186	2.195,28	42.898,6	44.683	53.476	3.176,7	50.406,4	17,5
	16	51.080	70.506	6.739,86	60.053,6	63.015	79.070	5.769,04	69.863,4	16,34
	32	68.314	105.250	13.476,34	79.895,4	75.055	149.058	28.329,47	104.042,4	30,22
	64	75.415	121.965	18.454,24	100.894,2	102.855	137.536	13.537,43	110.508,4	9,53
	128	82.834	128.586	15.765,86	99.354,6	112.733	178.248	30.909,71	151.170,2	52,15
	256	81.681	123.342	19.471,45	98.737,6	110.329	131.686	7.782,35	119.775,6	21,31
	512	83.520	117.870	13.307,04	93.962,4	130.031	155.237	11.832,71	144.488,8	53,77
	Média	**58.265,33**	**84.290,44**	**10.152,04**	**68.851,16**	**76.917,67**	**104.543,89**	**11.386,85**	**89.456,22**	29,93

Tabela A.22: Níveis de vazão para escrita sequencial com 2 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
2	1	21.963	22.014	16,31	21.985,8	29.771	30.037	103,8	29.846	35,75
	4	96.305	96.934	220,5	96.663,2	125.849	126.987	419,6	126.661	31,03
	8	111.709	113.253	490,81	112.464,8	182.255	183.488	456,71	182.824	62,56
	16	136.865	138.480	531,53	137.663	239.109	240.278	427,03	239.751,2	74,16
	32	142.443	143.117	222,57	142.800,4	271.183	276.231	1.988,1	274.184,8	92,01
	64	146.421	148.048	531,91	147.361,6	230.896	243.779	4.300,97	236.384,6	60,41
	128	145.905	150.657	1.744,06	149.327,6	246.298	256.083	3.550,48	249.456,8	67,05
	256	138.676	152.320	4.925,94	148.218,8	257.846	267.312	3.714,07	261.777,2	76,62
	512	148.048	153.480	2.133,53	150.947,6	277.401	279.570	748,46	278.286,2	84,36
	Média	**120.926,11**	**124.255,89**	**1.201,91**	**123.048,09**	**206.734,22**	**211.529,44**	**1.745,47**	**208.796,87**	69,69

Tabela A.23: Níveis de vazão para escrita sequencial com 4 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
4	1	24.548	24.603	23,18	24.576,8	44.782	45.076	120,31	44.962,2	82,95
	4	112.315	112.944	243,7	112.621	183.638	185.042	539,66	184.436,4	63,77
	8	132.062	132.362	109,61	132.195,2	296.040	298.286	869,26	297.296	124,89
	16	142.057	142.314	102	142.208,6	429.275	434.973	1.965,41	431.639,6	203,53
	32	142.224	143.208	335,19	142.839,6	554.606	559.738	1.858,27	556.456,8	289,57
	64	143.352	143.982	250,77	143.666,8	651.289	657.826	2.446,27	654.387	355,49
	128	140.836	142.922	698,55	142.048	703.427	713.640	3.930,83	711.274	400,73
	256	138.761	141.686	1.014,09	140.025,8	715.263	725.490	3.618,61	722.243,8	415,79
	512	137.284	139.425	840,96	138.091,6	712.993	726.496	4.613,37	721.261,8	422,31
	Média	**123.715,44**	**124.827,33**	**402**	**124.252,6**	**476.812,56**	**482.951,89**	**2.218**	**480.439,73**	286,66

Tabela A.24: Níveis de vazão para escrita sequencial com 8 processos.

Num. Procs.	Tam. Req. (KiB)	Vazão (KiB/s)								Variação Vazão CTR/CBC (%)
		CBC				CTR				
		Mínimo	Máximo	Desvio	Média	Mínimo	Máximo	Desvio	Média	
8	1	24.453	24.465	3,93	24.459,4	49.781	49.897	43,31	49.825,8	103,71
	4	111.943	112.191	104,99	112.107,8	203.028	203.448	143,11	203.196,2	81,25
	8	131.318	131.653	137,28	131.488	322.044	324.636	969,31	323.683,6	146,17
	16	140.603	140.950	120,26	140.766,8	459.297	462.471	1.146,79	461.158,4	227,6
	32	142.566	143.007	142,76	142.752,4	584.056	586.670	925,75	585.187,2	309,93
	64	141.514	142.379	296,99	141.882,4	664.075	672.307	2.960,07	669.344,8	371,76
	128	137.777	138.914	398,25	138.233	716.240	726.663	3.723,39	720.227,4	421,02
	256	135.550	136.373	345,34	135.867,2	722.823	731.734	2.915,25	727.919,4	435,76
	512	134.117	136.137	743,37	134.697,8	712.509	725.658	4.553,87	720.138,4	434,63
	Média	**122.204,56**	**122.896,56**	**254,8**	**122.472,76**	**492.650,33**	**498.164,89**	**1.931,21**	**495.631,24**	**304,69**

Apêndice B

WAESlib API

Este apêndice contém uma breve descrição das funções disponíveis na API da biblioteca WAESlib versão 2.01g0.

B.1 Função WAES_init()

```
int WAES_init(unsigned max_contexts, int n_pinned_pages,
unsigned long options);
```

Inicializa a biblioteca WAESlib, alocando memória na GPU para criar um *cache* com o numero máximo de contextos `max_contexts`. Cria os contextos de 0 a `(max_contexts-1)`.

- `n_pinned_pages`: deve indicar o número de segmentos de 4 KiB alocados em memória presa da CPU, para transferências CPU-GPU. Este número deve ser múltiplo de 32. A WAESlib usará no mínimo 32 páginas para esse propósito;
- `options`: define algumas opções que podem ser concatenadas e determinam, por exemplo, se a biblioteca CUDA deve ser inicializada ou não, tamanhos dos segmentos, etc. As opções disponíveis estão descritas no arquivo de cabeçalho da biblioteca (waeslib.h).

B.2 Função WAES_setKey()

```
int WAES_setKey(unsigned key_number, unsigned char key[],
unsigned cypher);
```

Configura uma chave de cifragem, a qual pode ser utilizada nas demais funções que exigem a utilização de uma chave.

- `key_number`: é um número de identificação que será atribuído à chave e será utilizado para referenciar a mesma nas demais funções;
- `key`: é um *buffer* onde está armazenada a chave a ser configurada;
- `cypher`: determina o cifrador a ser utilizado, bem como o tamanho da chave. Por exemplo, para utilizar o AES com chave de 128 bits, deve-se informar AES_CTR_128. Os valores possíveis também estão definidos no arquivo de cabeçalho da biblioteca.

B.3 Função WAES_ctx()

```
int WAES_ctx(unsigned ctx_number, unsigned key_number,
unsigned char IV[], unsigned priority);
```

Realiza a definição ou redefinição de um contexto de cifragem, informando a WAESlib que a mesma deve iniciar a produção da máscara de cifragem na GPU. Quando a produção da máscara termina, ela é transferida para a CPU, ficando disponível para utilização nas funções `WAES_encrypt()` e `WAES_decrypt()`.

- `ctx_number`: é o número de identificação do contexto, devendo ser algo entre 0 e o valor de `(max_contents-1)` definido na função `WAES_init()`;
- `key_number`: é o numero de identificação da chave de cifragem a ser utilizada, devendo corresponder a uma chave previamente configurada através da função `WAES_setKey()`;
- `IV`: *buffer* contendo o vetor de inicialização (*nonce*) a ser utilizado na geração da máscara;
- `priority`: prioridade utilizada para determinar a ordem de produção da máscara. Deve ser um número entre 0 e 63. Quanto menor o número, maior a prioridade.

B.4 Funções WAES_encrypt() e WAES_decrypt()

```
int WAES_encrypt(int ctx_number, unsigned char *buffer,
int n_bytes);
int WAES_decrypt(int ctx_number, unsigned char *buffer,
int n_bytes);
```

Fazem a cifragem ou decifragem, respectivamente, do *buffer* informado, aplicando a máscara gerada por um contexto previamente definido através da função `WAES_ctx()`. O processo é realizado em cima do próprio *buffer*.

A aplicação da máscara de cifragem é feita exclusivamente na CPU, consistindo de uma operação de `XOR` entre o *buffer* e a máscara produzida. O `XOR` é realizado utilizando instruções vetoriais SSE ou AVX pela WAESlib. Caso a produção da máscara do contexto referenciado já tenha finalizado na GPU, o processo se inicia imediatamente. Caso contrário, a função aguarda o término da produção da máscara, retornando à aplicação somente após a aplicação completa da máscara e consequentemente do processo de cifragem ou decifragem.

Contextos necessários a cifragens, cuja produção de máscaras foram atrasadas por terem sido declarados com menor prioridade, se tornam urgentes. Neste caso esses contextos são mudados automaticamente pela WAESlib para a maior prioridade (prioridade 0), consequentemente iniciando a produção de suas máscaras o quanto antes, a depender dos recursos para processamento paralelo disponíveis na GPU.

- `ctx_number`: é o número de identificação de um contexto previamente definido através da função `WAES_ctx()`;
- `buffer`: *buffer* contendo os dados a serem cifrados ou decifrados;

- `n_bytes`: tamanho do *buffer*, em bytes, a ser cifrado ou decifrado. Não deve ultrapassar o tamanho dos segmentos informado na função `WAES_init()`.

B.5 Função WAES_finish()

```
int WAES_finish(void);
```

Finaliza a biblioteca WAESlib, liberando memória e recursos alocados tanto na CPU quanto GPU. Também finaliza a biblioteca CUDA caso esta tenha sido inicializada pela WAESlib.

Apêndice C

Medições de vazão e utilização de CPU

Este apêndice contém tabelas com valores de vazão e utilização de CPU obtidos nas análises de desempenho discutidas nas Subseções 6.5.1 e 6.5.2.

C.1 Com o SAB armazenado em disco SSD

Tabela C.1: Níveis de vazão e utilização de CPU em leitura sequencial.

Versão		Tam. Req. (KiB)	Vazão (KiB/s)				Utilização CPU (%)			
			Mínimo	Máximo	Des. padrão	Média	Mínimo	Máximo	Des. padrão	Média
CBC	(S/AESNI)	4	224.555,00	291.171,00	28.894,78	271.947,00	9,41	12,10	1,19	11,32
		64	286.500,00	291.133,00	1.366,58	290.076,90	12,10	12,11	0,00	12,10
		128	288.979,00	291.095,00	652,71	290.709,90	12,10	12,13	0,01	12,11
	(AESNI)	4	366.498,00	444.624,00	35.731,43	408.063,80	9,51	11,47	0,88	10,54
		64	443.033,00	445.094,00	605,38	444.789,00	5,60	5,85	0,09	5,69
		128	444.198,00	445.009,00	264,11	444.627,40	5,32	5,62	0,09	5,48
CTR	(S/AESNI)	4	223.631,00	265.896,00	18.213,23	246.701,90	10,21	12,08	0,84	11,26
		64	262.250,00	265.723,00	1.333,75	264.562,10	12,07	12,12	0,01	12,08
		128	259.498,00	265.718,00	2.122,91	264.122,80	12,07	12,09	0,01	12,08
	(AESNI)	4	395.117,00	436.177,00	13.014,01	404.463,70	10,04	11,30	0,36	10,42
		64	442.739,00	443.763,00	372,00	443.480,40	5,49	5,67	0,06	5,57
		128	443.029,00	443.635,00	173,74	443.436,20	5,06	5,28	0,06	5,18
EncFS++		4	384.451,00	442.608,00	22.551,97	409.819,40	14,01	15,86	0,58	14,71
		64	442.679,00	443.132,00	128,59	443.007,10	11,20	12,20	0,30	11,54
		128	442.816,00	448.619,00	1.688,96	443.559,20	11,06	14,23	0,90	11,54

Tabela C.2: Níveis de vazão e utilização de CPU em escrita sequencial.

Versão		Tam. Req. (KiB)	Vazão (KiB/s)				Utilização CPU (%)			
			Mínimo	Máximo	Des. padrão	Média	Mínimo	Máximo	Des. padrão	Média
CBC	(S/AESNI)	4	92.101,00	96.199,00	1.075,18	95.146,80	10,23	10,48	0,07	10,41
		64	198.518,00	199.462,00	267,33	199.012,10	10,51	10,65	0,05	10,55
		128	192.682,00	201.069,00	2.155,28	195.190,30	9,78	10,32	0,14	9,93
	(AESNI)	4	124.548,00	128.865,00	1.162,86	127.551,00	9,66	9,86	0,06	9,79
		64	393.314,00	408.960,00	4.833,98	405.461,60	8,28	8,77	0,14	8,37
		128	435.455,00	444.877,00	3.227,78	442.628,50	8,18	8,50	0,09	8,30
CTR	(S/AESNI)	4	91.827,00	97.460,00	1.978,46	95.702,20	10,13	10,50	0,12	10,37
		64	216.424,00	221.171,00	1.455,93	219.947,20	10,26	10,45	0,05	10,35
		128	214.388,00	230.001,00	4.219,18	217.895,90	9,61	10,15	0,15	9,74
	(AESNI)	4	129.469,00	134.450,00	1.381,25	133.028,10	9,50	9,73	0,06	9,66
		64	553.046,00	582.866,00	10.301,59	573.651,60	7,22	7,50	0,10	7,43
		128	642.903,00	664.918,00	7.519,00	657.295,00	5,56	7,25	0,73	6,30
EncFS++		4	162.017,00	180.074,00	4.899,68	175.647,40	11,54	11,92	0,12	11,75
		64	621.194,00	657.002,00	11.971,62	635.454,40	13,68	14,68	0,34	14,16
		128	705.162,00	730.714,00	8.550,79	721.119,30	11,51	12,25	0,20	11,75

Tabela C.3: Níveis de vazão e utilização de CPU em leitura aleatória.

Versão		Tam. Req. (KiB)	Vazão (KiB/s)				Utilização CPU (%)			
			Mínimo	Máximo	Des. padrão	Média	Mínimo	Máximo	Des. padrão	Média
CBC	(S/AESNI)	4	17.259,25	17.552,50	90,69	17.381,99	1,89	1,99	0,03	1,93
		64	93.489,00	95.953,00	753,14	94.221,50	4,26	4,55	0,09	4,37
		128	116.069,00	118.389,00	763,44	117.357,70	5,24	5,44	0,06	5,34
	(AESNI)	4	17.056,00	17.539,00	151,59	17.336,30	1,31	1,40	0,02	1,35
		64	109.603,00	111.752,00	665,52	110.513,80	1,76	2,05	0,10	1,92
		128	140.334,00	144.890,00	1.518,76	142.500,70	2,06	2,32	0,08	2,19
CTR	(S/AESNI)	4	16.299,00	16.649,00	105,94	16.488,30	1,82	1,94	0,04	1,87
		64	63.150,00	91.251,00	8.774,13	86.588,60	4,32	7,10	1,04	4,95
		128	111.396,00	115.000,00	929,67	113.157,60	5,31	5,56	0,07	5,44
	(AESNI)	4	14.545,00	14.810,00	79,92	14.675,50	1,20	1,34	0,04	1,26
		64	108.750,00	113.988,00	1.359,24	111.722,60	1,74	2,11	0,10	1,88
		128	140.540,00	147.978,00	2.312,95	143.972,80	2,07	2,28	0,06	2,15
EncFS++		4	18.008,00	18.552,00	172,70	18.274,90	3,59	3,90	0,10	3,75
		64	103.135,00	107.822,00	1.440,06	105.415,50	5,11	5,65	0,18	5,35
		128	137.050,00	141.070,00	1.297,99	139.155,80	5,16	5,85	0,25	5,38

Tabela C.4: Níveis de vazão e utilização de CPU em escrita aleatória.

Versão		Tam. Req. (KiB)	Vazão (KiB/s)				Utilização CPU (%)			
			Mínimo	Máximo	Des. padrão	Média	Mínimo	Máximo	Des. padrão	Média
CBC	(S/AESNI)	4	93.731,00	94.782,00	348,67	94.346,50	10,32	10,42	0,03	10,36
		64	194.216,00	198.631,00	1.309,39	197.781,90	10,37	10,62	0,06	10,50
		128	191.871,00	196.289,00	1.082,24	194.243,90	9,75	10,02	0,08	9,91
	(AESNI)	4	118.483,00	126.946,00	2.433,67	125.473,70	9,47	9,71	0,07	9,64
		64	386.073,00	405.011,00	7.222,88	399.925,60	8,28	8,85	0,21	8,64
		128	436.724,00	439.102,00	713,06	437.508,90	8,21	8,52	0,12	8,32
CTR	(S/AESNI)	4	93.422,00	95.804,00	778,42	95.020,80	10,23	10,39	0,05	10,32
		64	215.667,00	219.919,00	1.221,44	218.925,00	10,21	10,42	0,06	10,32
		128	214.608,00	229.096,00	5.348,65	220.347,30	9,58	10,23	0,22	9,87
	(AESNI)	4	126.946,00	133.508,00	2.089,46	131.463,40	9,41	9,65	0,07	9,55
		64	557.753,00	573.305,00	5.238,11	568.135,70	7,37	7,65	0,08	7,46
		128	638.985,00	661.145,00	5.960,89	656.317,20	5,69	7,15	0,61	6,70
EncFS++		4	166.361,00	171.056,00	1.190,99	169.392,70	11,45	11,79	0,10	11,54
		64	607.517,00	647.668,00	10.430,77	625.814,20	13,62	14,50	0,25	14,06
		128	703.270,00	731.224,00	9.057,17	721.976,20	11,26	12,12	0,24	11,60

C.2 Com o SAB armazenado em memória

Tabela C.5: Níveis de vazão e utilização de CPU em leitura sequencial.

Versão		Tam. Req. (KiB)	Vazão (KiB/s)				Utilização CPU (%)			
			Mínimo	Máximo	Des. padrão	Média	Mínimo	Máximo	Des. padrão	Média
CBC	(S/AESNI)	4	315.284,00	316.051,00	186,48	315.673,40	12,19	12,23	0,01	12,20
		64	314.291,00	316.032,00	477,07	315.344,20	12,18	12,24	0,01	12,20
		128	314.022,00	315.951,00	563,74	315.456,70	12,19	12,23	0,01	12,20
	(AESNI)	4	529.047,00	535.464,00	1.737,40	532.391,00	12,19	12,20	0,00	12,20
		64	1.216.269,00	1.227.661,00	3.349,96	1.223.185,60	12,16	12,18	0,01	12,17
		128	1.294.238,00	1.303.894,00	2.842,05	1.299.305,90	12,13	12,20	0,02	12,15
CTR	(S/AESNI)	4	279.908,00	282.109,00	679,97	281.280,50	12,19	12,20	0,00	12,19
		64	277.336,00	282.199,00	1.377,64	281.257,30	12,18	12,24	0,02	12,20
		128	278.287,00	282.289,00	1.104,04	281.521,80	12,18	12,23	0,01	12,20
	(AESNI)	4	515.019,00	518.654,00	947,23	516.515,50	12,18	12,22	0,01	12,20
		64	1.245.617,00	1.275.445,00	8.353,09	1.255.543,00	12,13	12,16	0,01	12,15
		128	1.307.247,00	1.354.749,00	16.854,10	1.340.740,80	12,10	12,15	0,02	12,12
EncFS++		4	691.255,00	702.057,00	3.574,52	698.068,30	21,58	22,09	0,15	21,76
		64	1.194.192,00	1.380.954,00	50.060,61	1.323.435,30	23,65	24,01	0,11	23,90
		128	1.423.003,00	1.473.883,00	15.070,85	1.450.770,10	23,25	24,06	0,23	23,90

Tabela C.6: Níveis de vazão e utilização de CPU em escrita sequencial.

Versão		Tam. Req. (KiB)	Vazão (KiB/s)				Utilização CPU (%)			
			Mínimo	Máximo	Des. padrão	Média	Mínimo	Máximo	Des. padrão	Média
CBC	(S/AESNI)	4	99.922,00	101.089,00	362,19	100.473,60	10,37	10,44	0,02	10,42
		64	208.737,00	210.409,00	605,71	209.676,50	10,96	11,05	0,03	11,00
		128	204.760,00	215.179,00	2.906,50	206.597,40	10,33	10,96	0,18	10,43
	(AESNI)	4	135.672,00	137.515,00	522,52	136.500,80	9,79	9,88	0,03	9,82
		64	440.909,00	458.896,00	4.782,66	455.016,00	9,50	9,70	0,06	9,66
		128	497.152,00	499.781,00	777,79	498.829,30	9,65	9,69	0,01	9,66
CTR	(S/AESNI)	4	96.933,00	101.583,00	1.401,12	100.370,90	10,28	10,52	0,07	10,45
		64	232.752,00	234.732,00	573,67	234.054,10	10,87	10,94	0,02	10,89
		128	227.466,00	242.484,00	4.015,68	231.036,10	10,08	10,84	0,20	10,24
	(AESNI)	4	137.637,00	143.608,00	1.645,06	142.324,10	9,67	9,82	0,04	9,78
		64	671.381,00	687.518,00	4.657,20	685.164,60	8,45	8,50	0,01	8,49
		128	787.949,00	794.502,00	1.744,30	791.013,20	8,22	8,31	0,02	8,26
EncFS++		4	174.452,00	188.117,00	3.757,22	183.700,80	11,96	12,69	0,18	12,19
		64	770.133,00	795.671,00	7.184,80	788.011,50	17,60	18,78	0,35	18,59
		128	869.898,00	883.798,00	4.283,00	876.223,20	18,85	19,02	0,04	18,94

Tabela C.7: Níveis de vazão e utilização de CPU em leitura aleatória.

Versão		Tam. Req. (KiB)	Vazão (KiB/s)				Utilização CPU (%)			
			Mínimo	Máximo	Des. padrão	Média	Mínimo	Máximo	Des. padrão	Média
CBC	(S/AESNI)	4	161.361,00	163.679,00	736,79	162.777,60	10,33	10,46	0,03	10,37
		64	281.453,00	285.182,00	1.246,32	283.924,40	11,37	11,44	0,02	11,40
		128	287.938,00	292.153,00	1.516,75	290.148,60	11,33	11,54	0,06	11,45
	(AESNI)	4	265.330,00	277.246,00	3.787,79	274.526,80	9,07	9,20	0,04	9,15
		64	923.735,00	933.817,00	3.513,72	928.901,00	9,77	9,85	0,02	9,80
		128	1.012.898,00	1.025.587,00	3.320,90	1.016.858,50	9,86	9,91	0,02	9,89
CTR	(S/AESNI)	4	149.436,00	152.879,00	986,48	151.669,50	10,47	10,60	0,04	10,56
		64	253.836,00	257.604,00	1.142,72	255.632,50	11,40	11,51	0,03	11,45
		128	260.667,00	263.952,00	873,71	262.777,10	11,42	11,59	0,05	11,52
	(AESNI)	4	258.832,00	267.599,00	2.610,01	264.138,30	9,23	9,42	0,05	9,33
		64	930.246,00	951.663,00	5.597,25	938.167,50	9,80	9,82	0,01	9,81
		128	1.021.457,00	1.049.803,00	9.448,91	1.042.024,20	9,79	9,88	0,03	9,83
EncFS++		4	58.037,00	58.573,00	177,84	58.321,20	7,91	8,40	0,13	8,30
		64	518.147,00	533.412,00	4.042,01	528.848,20	14,94	17,50	0,76	17,12
		128	653.643,00	682.535,00	6.993,58	668.070,60	15,71	18,46	1,12	17,37

Tabela C.8: Níveis de vazão e utilização de CPU em escrita aleatoria.

Versão		Tam. Req. (KiB)	Vazão (KiB/s)				Utilização CPU (%)			
			Mínimo	Máximo	Des. padrão	Média	Mínimo	Máximo	Des. padrão	Média
CBC	(S/AESNI)	4	96.908,00	97.877,00	329,78	97.472,20	10,25	10,30	0,02	10,28
		64	205.541,00	209.654,00	1.137,51	208.820,60	10,88	11,00	0,03	10,97
		128	204.598,00	212.653,00	2.646,38	206.515,40	10,32	10,81	0,16	10,43
	(AESNI)	4	127.453,00	128.787,00	494,94	128.172,60	9,71	9,81	0,03	9,75
		64	449.804,00	451.468,00	490,79	450.610,90	9,61	9,66	0,01	9,64
		128	495.267,00	496.205,00	321,65	495.720,70	9,63	9,67	0,01	9,64
CTR	(S/AESNI)	4	92.409,00	98.268,00	1.744,21	96.735,50	10,10	10,40	0,08	10,32
		64	230.674,00	233.396,00	819,06	232.653,90	10,80	10,90	0,02	10,87
		128	224.940,00	230.905,00	1.950,09	228.019,60	10,00	10,25	0,09	10,13
	(AESNI)	4	132.095,00	139.126,00	1.830,49	137.356,40	9,45	9,65	0,06	9,60
		64	672.748,00	676.136,00	1.142,40	674.399,30	8,44	8,49	0,02	8,46
		128	759.800,00	782.812,00	6.526,87	777.447,00	8,11	8,25	0,04	8,21
EncFS++		4	176.319,00	179.212,00	959,73	177.859,80	11,85	12,01	0,05	11,94
		64	763.520,00	792.070,00	7.734,08	782.998,80	18,20	19,00	0,18	18,57
		128	858.199,00	873.166,00	5.170,60	867.641,60	18,79	19,04	0,08	18,89

Apêndice D

Análise de vazão com cifradores mínimo e nulo

Este apêndice apresenta gráficos e tabelas comparando valores de vazão alcançados pelo EncFS++, bem como pelas versões utilizando os cifradores mínimo e nulo. Sua principal utilidade é demonstrar valores máximos de vazão possíveis de serem alcançados na prática, dada a arquitetura atual do SAC. No cenário analisado, o SAB estava armazenado em memória. A experiência realizada correspondente a este apêndice está descrita ao final da seção 6.5.2, bem como brevemente discutida na seção 6.6.

D.1 Leitura e escrita sequenciais

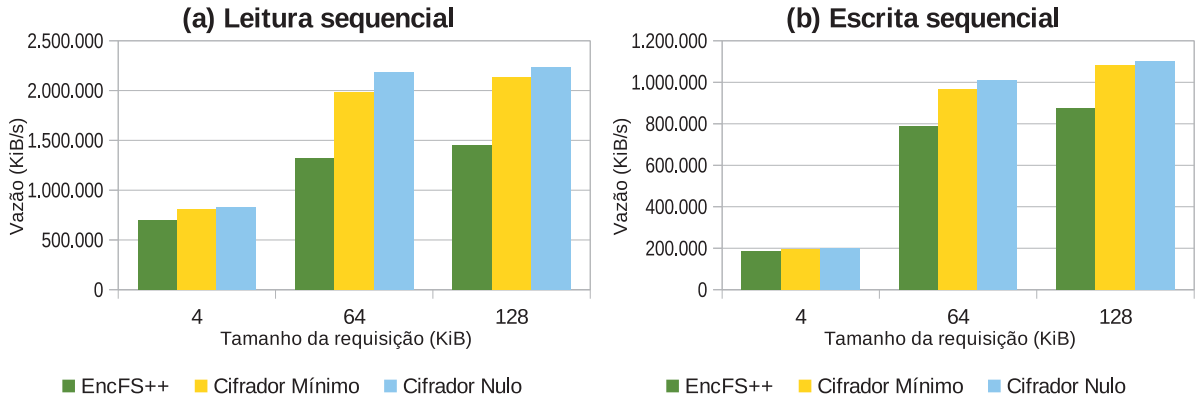


Figura D.1: Níveis de vazão em (a) leitura sequencial e (b) escrita sequencial.

Tabela D.1: Níveis de vazão em leitura sequencial.

Leitura Sequencial					
Tam. Req. (KiB)	Cifrador Mínimo	Cifrador Nulo	EncFS++	EncFS++ / Cif. Mín.	EncFS++ / Cif. Nulo
	Vazão (KiB/s)	Vazão (KiB/s)	Vazão (KiB/s)		
4	807.435,60	832.366,50	698.068,30	0,86	0,84
64	1.986.212,70	2.185.982,20	1.323.435,30	0,67	0,61
128	2.135.404,00	2.238.404,90	1.450.770,10	0,68	0,65
Média	**1.643.017,43**	**1.752.251,20**	**1.157.424,57**	**0,70**	**0,66**

Tabela D.2: Níveis de vazão em escrita sequencial.

Escrita Sequencial					
Tam. Req. (KiB)	Cifrador Mínimo	Cifrador Nulo	EncFS++	EncFS++ / Cif. Mín.	EncFS++ / Cif. Nulo
	Vazão (KiB/s)	Vazão (KiB/s)	Vazão (KiB/s)		
4	195.239,00	199.728,90	183.700,80	0,94	0,92
64	968.215,00	1.012.180,70	788.011,50	0,81	0,78
128	1.083.068,70	1.102.258,10	876.223,20	0,81	0,79
Média	**748.840,90**	**771.389,23**	**615.978,50**	**0,82**	**0,80**

D.2 Leitura e escrita aleatórias

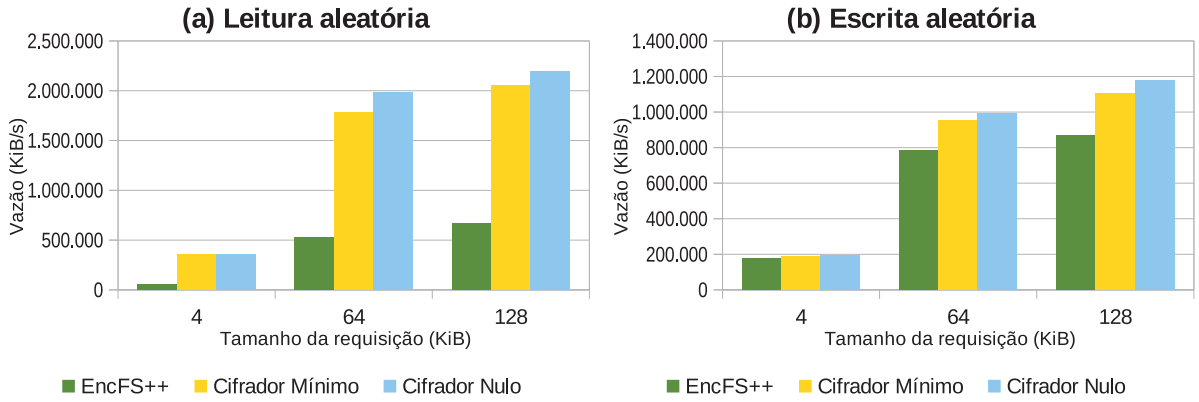


Figura D.2: Níveis de vazão em (a) leitura aleatória e (b) escrita aleatória.

Tabela D.3: Níveis de vazão em leitura aleatória.

Leitura Aleatória					
Tam. Req. (KiB)	Cifrador Mínimo	Cifrador Nulo	EncFS++	EncFS++ / Cif. Mín.	EncFS++ / Cif. Nulo
	Vazão (KiB/s)	Vazão (KiB/s)	Vazão (KiB/s)		
4	354.730,50	359.238,80	58.321,20	0,16	0,16
64	1.785.795,80	1.981.172,90	528.848,20	0,30	0,27
128	2.056.371,10	2.198.592,30	668.070,60	0,32	0,30
Média	**1.398.965,80**	**1.513.001,33**	**418.413,33**	**0,30**	**0,28**

Tabela D.4: Níveis de vazão em escrita aleatória.

Escrita Aleatória					
Tam. Req. (KiB)	Cifrador Mínimo	Cifrador Nulo	EncFS++	EncFS++ / Cif. Mín.	EncFS++ / Cif. Nulo
	Vazão (KiB/s)	Vazão (KiB/s)	Vazão (KiB/s)		
4	187.934,80	192.577,20	177.859,80	0,95	0,92
64	953.443,80	994.774,80	782.998,80	0,82	0,79
128	1.106.758,80	1.178.028,70	867.641,60	0,78	0,74
Média	**749.379,13**	**788.460,23**	**609.500,07**	**0,81**	**0,77**