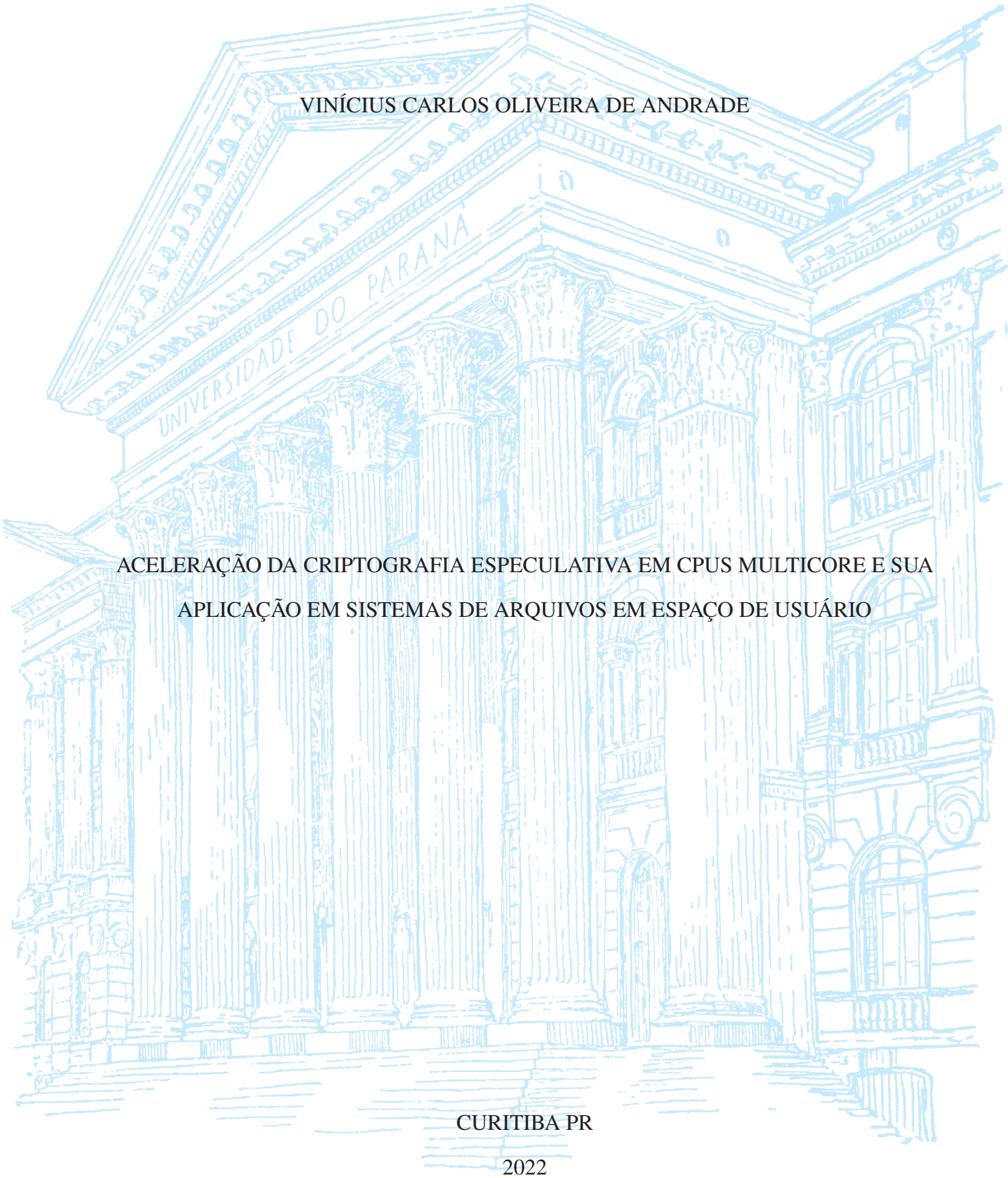
UNIVERSIDADE FEDERAL DO PARANÁ

VINÍCIUS CARLOS OLIVEIRA DE ANDRADE



ACELERAÇÃO DA CRIPTOGRAFIA ESPECULATIVA EM CPUS MULTICORE E SUA APLICAÇÃO EM SISTEMAS DE ARQUIVOS EM ESPAÇO DE USUÁRIO

CURITIBA PR

2022

VINÍCIUS CARLOS OLIVEIRA DE ANDRADE

ACELERAÇÃO DA CRIPTOGRAFIA ESPECULATIVA EM CPUS MULTICORE E SUA APLICAÇÃO EM SISTEMAS DE ARQUIVOS EM ESPAÇO DE USUÁRIO

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 M. Nunan Zola.

CURITIBA PR

2022

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Andrade, Vinícius Carlos Oliveira de
Aceleração da criptografia especulativa em CPUs Multicore e sua aplicação em sistemas de arquivos em espaço de usuário / Vinícius Carlos Oliveira de Andrade. – Curitiba, 2022.
1 recurso on-line : PDF.

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

Orientador: Wagner Machado Nunan Zola

1. Arquivos de computador. 2. Criptografia de dados (Computação). 3. Leitoras (Unidades de processamento de dados). I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Zola, Wagner Machado Nunan. IV. Título.

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **VINÍCIUS CARLOS OLIVEIRA DE ANDRADE** intitulada: **Aceleração da criptografia especulativa em CPUs multicore e sua aplicação em sistemas de arquivos em espaço de usuário**, sob orientação do Prof. Dr. WAGNER MACHADO NUNAN ZOLA, que após terem inquirido o aluno e realizada 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, 10 de Fevereiro de 2022.

Assinatura Eletrônica
11/02/2022 16:07:49.0
WAGNER MACHADO NUNAN ZOLA
Presidente da Banca Examinadora

Assinatura Eletrônica
11/02/2022 14:41:13.0
GERSON GERALDO HOMRICH CAVALHEIRO
Avaliador Externo (UNIVERSIDADE FEDERAL DE PELOTAS)

Assinatura Eletrônica
11/02/2022 13:32:21.0
DANIEL ALFONSO GONCALVES DE OLIVEIRA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Aos meus pais João e Valéria, que me permitiram estar onde estou hoje.
A minha irmã, que sempre me incentiva.
Aos meus gatos Sheldon e Leonard, que me atrapalham mas me alegram.
A minha esposa Gabriela, que sempre é minha amiga e companheira e está do meu lado em todos os momentos me dando força para seguir em frente. Te amo para sempre.

AGRADECIMENTOS

Primeiramente gostaria de agradecer aos meus pais João e Valéria. É impossível mensurar o quão importante foram para mim e para que eu pudesse chegar onde cheguei. Todo apoio, ajuda e incentivo me permitiram seguir em frente e me deram a oportunidade de fazer as escolhas que fiz. Muito obrigado por serem os melhores pais do mundo, sempre cuidarem de mim e me proporcionarem a vida que tenho.

Também gostaria de agradecer minha irmã Beatriz. Mesmo longe sempre me apoiava e se mostrava interessada no que estava fazendo. Muito obrigado por sempre me incentivar e apoiar e sempre ser a melhor irmã que alguém poderia ter.

Agradeço também ao professor Wagner. Seu apoio, orientação e paciência me guiaram na trilha que segui durante esse processo.

Aos meus gatos Sheldon e Leonard também agradeço por sempre me alegrarem nos momentos mais complicados.

Principalmente gostaria de agradecer a minha esposa Gabriela. Por muitos motivos esse período foi bastante complicado para nós. Só nós sabemos pelo que passamos e o que sentimos. Quero agradecer por sempre ser a melhor esposa do mundo e sempre estar ao meu lado. Desde muito antes, desde sempre, você sempre está lá por mim. Você sempre me dá a força que preciso para seguir em frente, sejam nos momentos felizes e nos não tão felizes. Obrigado por tudo. Obrigado por todos esses anos juntos. Obrigado por ser muito mais do que sempre sonhei. Sem você eu não seria metade do que sou hoje. De todo meu coração, muito obrigado.

A todos que me apoiaram neste caminho, muito obrigado!

RESUMO

Ao longo dos anos tem-se tornado cada vez mais aparente a importância com a segurança da informação. Enquanto antes a preocupação com os dados era amplamente focada em sua transmissão, percebe-se que a segurança dos dados armazenados também é vital em alguns sistemas. Apesar da preocupação as limitações das unidades de processamento e dos sistemas de armazenamento geravam dificuldades para a criação de um sistema que funcionasse em tempo real. Grandes esforços foram feitos nesta área de modo a buscar soluções viáveis neste cenário. Partes dos estudos focados em criptografia levaram a criação da WAESlib, trazendo como vantagem a utilização de especulação para melhorar a performance dos processos. Baseado nos processos criptográficos implementados na WAESlib foi criado o sistema de arquivos criptografado EncFS++ que apresenta uma excelente combinação de performance e segurança, porém necessitando de uma GPU para ser utilizado. Este trabalho visou adaptar a biblioteca WAESlib para gerar encriptação especulativa em CPU, ampliar o sistema de arquivos EncFS++ para ser utilizado em GPU ou CPU além de realizar testes de desempenho e de impacto durante sua utilização. Por ser mais adaptável para usos diversos, a versão do sistema de arquivos EncFS++ em CPU também foi submetido a testes em ambientes virtuais para que seja verificado se seu comportamento se mantém de acordo com o encontrado para os ambientes físicos. Os principais quesitos verificados foram a vazão e o impacto sofrido pelo sistema em termos de porcentagem de uso de CPU e de memória. Os testes foram realizados utilizando ferramenta que simulam a utilização do sistema de arquivos por diferentes tipos de aplicação, *Filebench*, buscante verificar o comportamento do sistema o mais próximo possível de um ambiente real. A primeira parte deste trabalho focou na adaptação da biblioteca WAESlib para que funcionasse em CPU e foram realizados testes preliminares que mostram que o comportamento está dentro do esperado para a implementação, tanto para as funções internas quanto para a vazão dos dados criptografados. Os resultados também são coerentes com alguns encontrados na literatura relativas a encriptação paralela em CPU com vários núcleos. A segunda parte teve seu foco na adaptação e testes do sistema de arquivos EncFS++ em diversos ambientes. Foi verificado o ganho de vazão e o comportamento do sistema de arquivos quando comparado com o sistema base EncFS. Os resultados da primeira e segunda etapa mostram a viabilidade da utilização do sistema de arquivos e da WAESlib com processamento em CPU, apresentando ganhos em diversos ambientes, incluindo ambientes virtualizados ou sem hardware específico para processamento de criptografia.

Palavras-chave: Sistema de arquivos. Criptografia especulativa. CPU. WAESlib. EncFS++.

ABSTRACT

Throughout the years the importance of information security has become increasingly apparent. Despite before the concern with data were mostly focused on during transmission it is being noticed that stored data security is also vital in some systems. Despite the concerns the limitations of the processing unities and storage systems created difficulties to the creation of a system that worked in real time, in spite of this major efforts were made in this area looking for viable solutions. Part of this studies focused in cryptography lead to the creations of WAESlib, that brought as advantages the use of speculation to improve the performance of the processes. Based on the cryptographic processes implemented on WAESlib the encrypted file ssystem EncFS++ was created that presents an excellent combination of performance and security, even though it needs a GPU to be able to be used. This work adapted the WAESlib library so that it can generate speculative encryption in CPU, expanded the EncFS++ file system to be able to be used in GPU or CPU and tested the performance and impact during its use. Being more easily adaptable for different uses the CPU version of the EncFS++ file system were also subjected to tests in virtual environments so that it is verified if its behavior stays in accordance to the one found for physical environments. The main verified items were throughput and the impact felt by the system in terms of percentage of use of CPU and memory. The tests were conducted using tools that simulate the use of the file system by different kinds of applications, Filebench, trying to verify the behavior of the system as close as possible to the real environment. The first part of this work focused on adaptating the WAESlib so that it worked on CPU and preliminary testes were done show that the behavior is within the expected for the implementation, not only to the for the internal functions but also for the encrypted data throughput. The results are also coherent with some other found in the literature relative to parallel encryption in multicore CPU. The second part focused on adapting and testing of the EncFS++ filesystem on multiple environments. A throughput gain and the filesystem behavior were verified when compared with the base sistem EncFS. The results os the first and second part show the viability of using the filesystem and the WAESlib with CPU processing, showing gains in multiple environments, including virtual environments or environments without specific hardware for processing cryptography.

Keywords: File system. Speculative cryptography. CPU. WAESlib. EncFS++.

LISTA DE FIGURAS

2.1 Modelo de encriptação e decriptação simétrico, baseado em [1]. 17
2.2 Algoritmo AES para encriptação e decriptação, baseado em [2]. 19
2.3 Função de transformação *SubBytes*, baseada em [3]. 20
2.4 Função de transformação *ShiftRows*, baseada em [3]. 21
2.5 Função de transformação *MixColumns*, baseada em [3]. 22
2.6 Função de transformação *AddRoundKey*, baseada em [3]. 23
2.7 Funcionamento do modo ECB, baseado em [2]. 26
2.8 Imagem antes e depois de ser encriptada utilizando modo ECB [4]. 27
2.9 Funcionamento do modo CBC, baseado em [2]. 28
2.10 Funcionamento do modo CFB, baseado em [2]. 29
2.11 Funcionamento do modo OFB, baseado em [2]. 31
2.12 Funcionamento do modo CTR, baseado em [2]. 32
2.13 Modelos de camadas de um sistema operacional Linux, baseado em [5]. 34
2.14 Crescimento do número de linhas de código do *kernel* do Linux ao longo dos anos [6]. 36
2.15 Composição de um *kernel*. 36
2.16 Componentes de um VFS, baseada em [7] 37
2.17 Exemplo de funcionamento de um sistema de arquivos utilizando FUSE [8]. . . . 39
2.18 Exemplo de instrução SIMD e escalar, baseado em [9]. 42

4.1 Ligações entre as diferentes partes da WAESlib CPU 49
4.2 Resultado do teste de requisição sequencial variando o número de *threads* clientes. 50
4.3 Resultado do teste de requisição aleatória variando o número de *threads* trabalhadoras. 51

5.1 Teste variando a quantidade de clientes em ambiente físico sobre dispositivo NVMe. 54
5.2 Teste variando a quantidade de clientes em ambiente físico sobre dispositivo SSD. 54
5.3 Teste variando a quantidade de clientes em ambiente virtual sobre dispositivo SSD. 55
5.4 Teste variando a quantidade de clientes em ambiente virtual sobre dispositivo HDD. 55
5.5 Teste variando a quantidade de threads trabalhadoras em ambiente físico sobre dispositivo NVMe e utilizando AES-NI. 57
5.6 Teste variando a quantidade de threads trabalhadoras em ambiente físico sobre dispositivo NVMe e sem utilizar AES-NI. 57
5.7 Teste variando a quantidade de threads trabalhadoras em ambiente físico sobre dispositivo SSD e utilizando AES-NI. 58

5.8 Teste variando a quantidade de threads trabalhadoras em ambiente físico sobre dispositivo SSD e sem utilizar AES-NI. 58
5.9 Teste variando a quantidade de threads trabalhadoras em ambiente virtual sobre dispositivo SSD e utilizando AES-NI. 59
5.10 Teste variando a quantidade de threads trabalhadoras em ambiente virtual sobre dispositivo SSD e sem utilizar AES-NI. 59
5.11 Teste variando a quantidade de threads trabalhadoras em ambiente virtual sobre dispositivo HDD e utilizando AES-NI. 60
5.12 Teste variando a quantidade de threads trabalhadoras em ambiente virtual sobre dispositivo HDD e sem utilizar AES-NI. 60

A.1 Comportamento dos testes realizados sobre o EncFS++ em ambiente físico sobre dispositivo NVMe e utilizando AES-NI. 68
A.2 Comportamento dos testes realizados sobre o EncFS em ambiente físico sobre dispositivo NVMe e utilizando AES-NI. 68
A.3 Comportamento dos testes realizados sobre o EncFS++ em ambiente físico sobre dispositivo NVMe e sem utilizar AES-NI. 69
A.4 Comportamento dos testes realizados sobre o EncFS em ambiente físico sobre dispositivo NVMe e sem utilizar AES-NI. 70
A.5 Comportamento dos testes realizados sobre o EncFS++ em ambiente físico sobre dispositivo SSD e utilizando AES-NI. 71
A.6 Comportamento dos testes realizados sobre o EncFS em ambiente físico sobre dispositivo SSD e utilizando AES-NI. 71
A.7 Comportamento dos testes realizados sobre o EncFS++ em ambiente físico sobre dispositivo SSD e sem utilizar AES-NI. 72
A.8 Comportamento dos testes realizados sobre o EncFS em ambiente físico sobre dispositivo SSD e sem utilizar AES-NI. 73
A.9 Comportamento dos testes realizados sobre o EncFS++ em ambiente virtual sobre dispositivo SSD e utilizando AES-NI. 74
A.10 Comportamento dos testes realizados sobre o EncFS em ambiente virtual sobre dispositivo SSD e utilizando AES-NI. 74
A.11 Comportamento dos testes realizados sobre o EncFS++ em ambiente virtual sobre dispositivo SSD e sem utilizar AES-NI. 75
A.12 Comportamento dos testes realizados sobre o EncFS em ambiente virtual sobre dispositivo SSD e sem utilizar AES-NI. 76
A.13 Comportamento dos testes realizados sobre o EncFS++ em ambiente virtual sobre dispositivo HDD e utilizando AES-NI. 77
A.14 Comportamento dos testes realizados sobre o EncFS em ambiente virtual sobre dispositivo HDD e utilizando AES-NI. 77
A.15 Comportamento dos testes realizados sobre o EncFS++ em ambiente virtual sobre dispositivo HDD e sem utilizar AES-NI. 78
A.16 Comportamento dos testes realizados sobre o EncFS em ambiente virtual sobre dispositivo HDD e sem utilizar AES-NI. 79

LISTA DE TABELAS

2.1 Tabela S-box direta conforme definida para o AES, extraída de [2]. 20
2.2 Tabela S-box inversa conforme definida para o AES, extraída de [2]. 21

A.1 Resultados dos testes de vazão (MB/s) realizados em ambiente físico sobre NVMe utilizando AES-NI. 67
A.2 Resultados dos testes de vazão (MB/s) realizados em ambiente físico sobre NVMe sem utilizar AES-NI. 69
A.3 Resultados dos testes de vazão (MB/s) realizados em ambiente físico sobre SSD utilizando AES-NI. 70
A.4 Resultados dos testes de vazão (MB/s) realizados em ambiente físico sobre SSD sem utilizar AES-NI. 72
A.5 Resultados dos testes de vazão (MB/s) realizados em ambiente virtual sobre SSD utilizando AES-NI. 73
A.6 Resultados dos testes de vazão (MB/s) realizados em ambiente virtual sobre SSD sem utilizar AES-NI. 75
A.7 Resultados dos testes de vazão (MB/s) realizados em ambiente virtual sobre HDD utilizando AES-NI. 76
A.8 Resultados dos testes de vazão (MB/s) realizados em ambiente virtual sobre HDD sem utilizar AES-NI. 78

LISTA DE ACRÔNIMOS

AES	Advanced Encryption Standard
NIST	National Institute of Standards and Technology
GPU	Graphics processing unit
WAES	Warped AES
CPU	Central processing unit
AVX	Advanced Vector Extensions
AES-NI	Advanced Encryption Standard New Instructions
CTR	Counter Mode
DES	Data Encryption Standard
XOR	Ou exclusivo
ECB	Electronic Codebook
CBC	Cipher Block Chaining
CFB	Cipher Feedback
OFB	Output Feedback
IV	Vetor de inicialização
RAM	Random Access Memory
VFS	Virtual File System
HD	Hard Disk
FUSE	Filesystem in Userspace
SIMD	single instruction multiple data
SSE	Streaming SIMD Extensions
WML	Workload model language

LISTA DE SÍMBOLOS

$\oplus$	Operação Ou Exclusivo (XOR)

SUMÁRIO

1 INTRODUÇÃO 14
2 FUNDAMENTAÇÃO TEÓRICA 16
2.1 ALGORITMOS DE CRIPTOGRAFIA SIMÉTRICOS E O AES 16
2.1.1 Cifragem simétrica 16
2.1.2 *Advanced Encryption Standard* 17
2.1.3 Modos de operação 24
2.2 ESTRUTURA DO LINUX E SISTEMAS DE ARQUIVO CRIPTOGRAFADOS 33
2.2.1 Estrutura do Linux 34
2.2.2 Kernel 35
2.2.3 VFS 37
2.2.4 FUSE 38
2.2.5 EncFS 38
2.2.6 Sistemas de arquivos em espaço de *kernel* 40
2.3 EXTENSÕES AVX E INSTRUÇÕES AES-NI 41
2.3.1 AVX 41
2.3.2 AES-NI 42
2.4 MÉTODOS DE AVALIAÇÃO PARA SISTEMAS DE ARQUIVOS 43
2.5 ASPECTOS PRINCIPAIS 44
3 TRABALHOS RELACIONADOS 45
4 IMPLEMENTAÇÃO E TESTES DA BIBLIOTECA WAESLIB CPU 47
4.1 WAESLIB 47
4.2 IMPLEMENTAÇÃO DA CRIPTOGRAFIA ESPECULATIVA EM CPU 48
5 IMPLEMENTAÇÃO E TESTES DO SISTEMA DE ARQUIVOS ENCFS++ EM CPU 52
5.1 AMBIENTES DE TESTE 52
5.2 EXPERIMENTOS VARIANDO O NÚMERO DE CLIENTES 53
5.3 EXPERIMENTO VARIANDO O NÚMERO DE THREADS TRABALHADORAS 55
6 CONCLUSÃO 61
6.1 RESULTADOS OBTIDOS 61
6.2 TRABALHOS FUTUROS 62
REFERÊNCIAS 63

APÊNDICE A – RESULTADOS E GRÁFICOS DOS TESTES DO ENCFS++. 67
A.1 SISTEMA FÍSICO COM DISPOSITIVO NVME 67
A.1.1 Resultados utilizando AES-NI 67
A.1.2 Resultados não utilizando AES-NI 69
A.2 SISTEMA FÍSICO COM DISPOSITIVO SSD. 70
A.2.1 Resultados utilizando AES-NI 70
A.2.2 Resultados não utilizando AES-NI 72
A.3 SISTEMA VIRTUAL COM DISPOSITIVO SSD 73
A.3.1 Resultados utilizando AES-NI 73
A.3.2 Resultados não utilizando AES-NI 75
A.4 SISTEMA VIRTUAL COM DISPOSITIVO HDD. 76
A.4.1 Resultados utilizando AES-NI 76
A.4.2 Resultados não utilizando AES-NI 78

1 INTRODUÇÃO

Desde a antiguidade as pessoas buscam maneiras de se assegurar que o conteúdo de uma mensagem sigilosa seja recuperado apenas por seu destinatário, um dos exemplos mais antigos registrados a chamada cifra de César [10], utilizada pelo imperador romano Júlio César já no século I a.C. Com o grande volume de dados sendo criados atualmente, com estimativas de que aproximadamente 50 zettabytes tenham sido criados em 2020 [11], esta demanda por métodos que, não somente tragam a confidencialidade desejada às informações, tanto no envio quanto no armazenamento, como também o façam de forma rápida e eficiente está cada vez maior e cresce ainda mais a cada dia.

Embora no início do desenvolvimento da computação grande parte dos sistemas não possuíssem nenhum tipo de segurança, tanto de acesso quanto de seus dados, logo notou-se que seriam necessárias medidas mais restritivas. Com a percepção dessa necessidade surgiu o que hoje é chamado de segurança da informação e, com ela, foram definidos três propriedades principais que um sistema deve ter para que possa ser considerado seguro, sendo eles a confidencialidade, que é a necessidade de que os dados só sejam acessados por seus destinatários, a integridade, que é a garantia que o dado não foi alterado, e a disponibilidades, que é a certeza que o dado poderá ser acessado quando for necessário [12].

Buscando seguir as políticas mínimas preconizadas pela segurança da informação foram desenvolvidas diversas técnicas computacionais de criptografia que, de forma geral, diferem significativamente das não computacionais, baseadas em substituições simples. Algumas dessas novas técnicas são as chamadas cifras simétricas. Este tipo de técnica é largamente utilizada para o armazenamento de dados pois, de maneira geral, apresenta alta vazão, chaves relativamente pequenas e pode ser concatenada para gerar cifras mais fortes, tornando cifras fracas em cifras utilizáveis [13].

Com o aumento do poder computacional dos processadores e a diminuição do tempo de escrita e leitura dos dispositivos de armazenamento, tornou-se possível a utilização de criptografia em tempo real dos arquivos armazenados em computadores. Com essa possibilidade em vista, surgiram os sistemas de arquivos criptográficos que, por padrão, criptografam todos os arquivos e diretórios existentes nele, não sendo necessário definir individualmente quais arquivos serão criptografados.

Apesar de terem surgido com o aumento do poder dos computadores, os sistemas de arquivos criptográficos ainda geram um número muito grande de requisições para o processador e um volume muito alto de bytes para serem lidos e escritos nos dispositivos de armazenamento, o que consome muitos recursos computacionais. Por esse motivos sempre buscou-se minimizar o número de requisições e acelerar o processamento das mesmas, sendo inicialmente criadas soluções sobre placas aceleradoras que, de forma geral, possuíam um alto custo e pouca flexibilidade [14].

A busca por melhor desempenho levou pesquisadores e desenvolvedores a procurarem melhores técnicas para o processamento de cifras criptográficas. O desenvolvimento do cifrador simétrico *Advanced Encryption Standard* (AES), em 1998 e sua posterior oficialização pelo *National Institute of Standards and Technology* (NIST), em 2001 [3], levaram a um grande número de pesquisas na área, principalmente utilizando processamento paralelo em *graphics processing unit* (GPUs). Uma dessas pesquisas [15] levou a criação da *Warped* AES (WAES) e da WAESlib, uma biblioteca para integrar o WAES com outros algoritmos de encriptação paralelos.

Uma das vantagens da WAES é sua capacidade de realizar a encriptação de dados de forma antecipada, deixando os dados prontos para quando forem requisitados. Embora a WAES tenha sido criada para funcionar em GPU sua característica de encriptar dados antecipadamente permite que seja adaptada para uso em *central processing unit* (CPU). O lançamento das *Advanced Vector Extensions* (AVX) para os processadores Intel no ano de 2008 e das *Advanced Encryption Standard New Instructions* (AES-NI) , que permite o processamento de AES de forma muito mais rápida, bem como o uso de múltiplos núcleos de processamento permitem encriptação antecipada de artefatos de forma paralela e eficiente. Embora os resultados esperados, em volume de dados criptografados, com o uso somente de CPU sejam menores que os em GPU estes possuem a vantagem de serem mais flexíveis e adaptáveis, por abrangerem um maior número de dispositivos em que podem ser utilizados até mesmo em máquinas virtuais sem a necessidade de alteração ou de GPU *passthrough*.

Para este trabalho foi implementada a biblioteca WAESlib em sua versão em CPU, dando continuidade ao trabalho desenvolvido anteriormente em GPU [14] e complementando o sistema de arquivos EncFS++, também na versão com processamento especulativo em CPU. Foram utilizadas as ferramentas oferecidas pela biblioteca Libcrypto, que é a parte do OpenSSL responsável por fornecer as rotinas fundamentais de criptografia, juntamente com a biblioteca *Pthread*, que cuida da criação e gerenciamento de *threads* paralelas em CPU, para o desenvolvimento do modo de criptografia especulativo da WAESlib. Para o EncFS++ foi utilizada a biblioteca WAESlib em CPU para sua implementação e otimizados parâmetros visando alcançar os melhores resultados nos testes realizados com o *Filebench* [16].

Este trabalho possuiu como objetivo averiguar a viabilidade da utilização das técnicas de WAES para aceleração da encriptação especulativa quando implementadas para realizarem processamento paralelo em CPUs multicore, além de verificar sua aplicação em sistemas de arquivos encriptados em espaço de usuário. Para alcançar os objetivos foram realizados diversos testes de simulação de utilização da WAESlib e do EncFS++ com processamento em CPU. Os testes realizados com a ferramenta *filebench* apresentaram resultados positivos, trazendo como contribuições para o trabalho não apenas as implementações da WAESlib e do EncFS++ versão CPU como também a demonstração de sua viabilidade.

Na Seção 2.1 será apresentado um pouco mais sobre métodos de criptografia. É falado um pouco mais sobre como funcionam métodos simétricos de criptografia, suas vantagens e desvantagens e serão apresentados alguns de seus modos de operação, analisando as características de cada um e mostrando porque o modo CTR foi escolhido. A Seção 2.2 apresentará um pouco mais sobre a estrutura e funcionamento interno do Linux e dos sistemas de arquivo em espaço de usuário, completando apresentando a EncFS, base do sistema EncFS++. A Seção 2.3 falará sobre as extensões AVX e as instruções AES-NI utilizadas para o aumento de performance para algumas operações em CPU. Para a avaliação de sistemas de arquivos são apresentados alguns métodos na Seção 2.4. O Capítulo 3 apresenta alguns trabalhos relacionados sobre criptografia e sistemas de arquivos. O Capítulo 4 apresenta detalhes sobre a implementação e testes iniciais realizados com o biblioteca WAESlib versão CPU. No Capítulo 5 são mostrados os resultados obtidos nos testes realizados utilizando o sistema de arquivos EncFS++ implementado com a biblioteca WAESlib CPU. No Capítulo 6 são realizadas algumas considerações finais sobre o trabalho, consolidando os resultados obtidos e apresentando possibilidades para trabalhos futuros. No Apêndices A são apresentados todos os resultados obtidos em cada um dos ambientes de teste, tanto para o EncFS quanto para o EncFS++.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta um pouco sobre as fundamentações teóricas relacionadas com o entendimento, implementação e testes da EncFS++ CPU. As seções a seguir apresentam tópicos sobre métodos de criptogra, estrutura do Linux, extensões AVX, instruções AES-NI e métodos para avaliação de sistemas de arquivos.

2.1 ALGORITMOS DE CRIPTOGRAFIA SIMÉTRICOS E O AES

O uso de criptografia é algo essencial para a sociedade contemporânea. O que uma vez foi algo utilizado majoritariamente apenas por bancos, governos e militares hoje é base de funcionamento dos mais diversos sistemas computacionais. Mesmo alguém que não possua nenhum conhecimento na área de criptografia, ou que nem mesmo já tenha ouvido sobre o conceito, certamente já se utilizou sistemas que usam a criptografia, como navegadores web, sistemas de transmissão de mensagens, sistemas bancários, entre outros e se aproveitou de suas funcionalidades.

Conforme definido no livro *Handbook of Applied Cryptography* [13] criptografia é o estudo de técnicas matemáticas relacionadas a segurança da informação, completando que criptografia não é um meio de prover segurança da informação, e sim um conjunto de técnicas. As técnicas criptográficas buscam trazer confidencialidade, garantindo que a mensagem só seja lida por seu destinatário, integridade, sendo a garantia de que a mensagem não foi alterada, autenticidade, que é a detecção das partes envolvidas na criação da mensagem e irretratabilidade, impossibilitando as partes de negar que acessaram a mensagem [14].

Cifragens são conjuntos de técnicas, chamadas de métodos criptográficos, que visam tornar uma mensagem incompreensível para um leitor não autorizado, enquanto decifragem são as técnicas utilizadas para recuperação da mensagem original. Os métodos criptográficos são divididos em duas partes, sendo elas cifragem simétrica e assimétrica, cada uma possuindo vantagens e desvantagens sobre a outra e podendo operar em diversos modos diferentes.

Na Subseção 2.1.1 será apresentado um pouco mais sobre o funcionamento dos algoritmos de criptografia simétricos, na Subseção 2.1.2 será mostrado o algoritmo de criptografia AES e na Subseção 2.1.3 serão apresentados os modos de utilização do AES.

2.1.1 Cifragem simétrica

Cifragens simétricas são aquelas em que se pode conseguir a chave para decriptação a partir da chave de encriptação por um método computacionalmente fácil [13]. De maneira geral a grande maioria das cifragens simétricas utilizadas fazem uso da mesma chave tanto para cifrar quanto para decifrar mensagens. O processo de cifragem e recuperação das mensagens é geralmente representada de maneira simplificada por um diagrama como o da Figura 2.1. Os termos texto em claro ou texto normal e texto cifrado ou cifra são largamente utilizado para se referirem à mensagem antes de passar pelo processo de encriptação e depois do processo. O termo chave ou chave secreta é utilizado para indicar o valor que será utilizado para cifrar ou decifrar a mensagem.

Uma mensagem inicialmente não criptografada é processada pelo algoritmo de criptografia utilizando uma chave e, com isso, gerando um texto cifrado. O texto cifrado por sua vez deve ser processado utilizando o algoritmo de decifragem, que faz o processo inversos do de

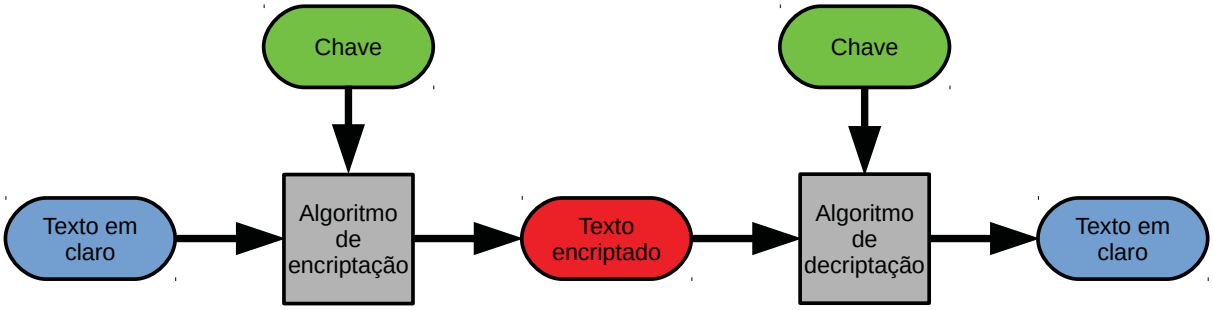


Figura 2.1: Modelo de encriptação e decriptação simétrico, baseado em [1].

cifragem, para que possa ser lido pelo destinatário da mensagem. É possível que os algoritmos de cifragem e decifragem sejam os mesmos. Os algoritmos de criptografia simétricos podem ser divididos em duas categorias, sendo cifra de bloco ou cifras de fluxo.

Cifras de blocos são aquelas que dividem a mensagem em claro em diversas partes de tamanho fixo, chamados de blocos, e as encripta um bloco de cada vez [13]. O bloco de saída, contendo a cifra, possui o mesmo tamanho do bloco de entrada, contendo o texto em claro. A maior parte das cifras simétricas largamente utilizadas, como o AES por exemplo, são cifras de bloco em que cada bloco possui tamanho 64 ou 128 bits.

Cifra de fluxo por sua vez podem ser consideradas cifras de blocos muito simples, onde cada bloco tem tamanho um. Uma de suas grandes vantagens é o fato dos erros não se propagarem pela mensagem. Uma vez que seu bloco possui tamanho um os erros são isolados apenas no símbolo em que ele ocorreu, deixando o restante da mensagem intacta. Outra vantagem é nos casos em que os símbolos devem ser processados um de cada vez, em caso de limitação de hardware por exemplo [13].

As cifras simétricas possuem, de maneira geral, como vantagens alta vazão durante a encriptação e decriptação de mensagens e chaves relativamente curtas quando comparadas com algoritmos assimétricos. Em contrapartida possuem as desvantagens de que as partes envolvidas com a encriptação e decriptação devem concordar com qual chave utilizar, o que deve ser feito de maneira prévia e, em ambientes com muitas partes envolvidas, o gerenciamento das chaves pode se tornar algo complexo [13]. Essas características indicam que as cifras simétricas são ideais para serem utilizadas em sistemas de armazenamento, uma vez que a alta vazão permite a escrita de dados com pouca perda de desempenho, as chaves pequenas ocupam pouco espaço e o gerenciamento das chaves se torna mais simples pois, de maneira geral, quem armazena a mensagem é o mesmo que a recupera posteriormente.

2.1.2 *Advanced Encryption Standard*

O AES é um algoritmo de cifragem simétrico em blocos desenvolvida em 1998 por Joan Daemen e Vincent Rijmen e posteriormente aceita pelo NIST em 2001 [3]. O concurso do NIST no qual o AES foi introduzido buscava um substituto para o *Data Encryption Standard* (DES) [17] e o 3DES [18] uma vez que esses já se mostravam inadequados do ponto de vista de segurança e ineficientes quando implementados em software. O NIST exigiu que todos os algoritmos participantes fossem eficientes tanto em hardware quanto em software, possuíssem blocos de 128 bits e chaves de pelo menos 128 bit. Seguindo as exigências do NIST o AES possui três versões de tamanho padrão de chaves, 128, 192 e 256 bits, e para todos os modos o tamanho do bloco é sempre 128 bits, diferente dos 64 bits utilizados pelo DES.

O algoritmo para cifragem utilizado pelo AES pode ser subdividido em algumas partes, sendo elas duas entradas, o bloco a ser criptografado e a chave, quatro funções, chamadas

SubBytes, *ShiftRows*, *MixColumns* e *AddRoundKey*, o *key schedule*, responsável por gerar as chaves de cada rodada, *round*, e um bloco de saída. O tamanho da chave dita quantas rodadas do algoritmo serão executadas antes de ser finalizado, sendo o número de rodadas denominado Nr e possuindo valores 10, 12 e 14 para chaves de tamanho 128, 192 e 256 respectivamente [3].

O algoritmo AES se inicia com a definição do número de rodadas que serão executados e, com essa informação, o *key schedule* expande a chave para um tamanho total equivalente a 11, 13 ou 15 chaves, sendo igual ao número de rodadas mais um, $Nr + 1$. Cada uma dessas chaves será utilizada em uma rodada do algoritmo. A primeira chave é utilizada antes de se iniciarem as rodadas de processamento sendo a aplicação da função *AddRoundKey* do estado inicial com a primeira chave, que significa realizar um ou exclusivo (XOR) entre eles. As próximos rodadas, com exceção do último, são iguais e compostas por uma aplicação sequencial de cada uma das funções do AES, sendo a ordem de aplicação *SubBytes*, *ShiftRows*, *MixColumns* e *AddRoundKey*. A última rodada é similar às anteriores porém com a pequena diferença de que nele a função *MixColumns* não é aplicada, somente aplicando as funções *SubBytes*, *ShiftRows* e *AddRoundKey* nessa ordem. A função de decriptação segue um processo similar, porém as chaves de rodada são utilizadas na ordem inversa e as funções sofrem pequenas alterações. A Figura 2.2 ilustra o funcionamento do algoritmo AES com chave de 128 bits. A Figura 2.2 (a), na esquerda, mostra o processo de encriptação do bloco, de modo que seguindo de cima para baixo o texto claro, é transformado em uma cifra e Figura 2.2 (b), na direita, mostra o processo de decriptação no qual seguindo debaixo para cima transforma o texto cifrado em texto claro. As chaves de rodada são representadas como partes de um vetor w que é utilizado para armazenar as chaves. O vetor w é dividido em elementos de 32 bits, enquanto uma chave possui 128 bit, mostrando porque em cada rodada são escolhidos quatro elementos.

Os resultados parciais de cada rodada são chamados de estados. Um estado pode ser visto como uma matriz com quatro linhas e o número de colunas é definido como o tamanho de um bloco dividido por 32, dessa maneira cada elemento possui 8 bits, 1 byte [3] e, como os blocos do AES possuem 128 bits as matrizes possuem dimensões 4 X 4. Da mesma forma como se divide o estado também é possível se dividir a chave, em matrizes com quatro linhas, porém é utilizado o termo Nk para o número de colunas para que se possa fazer distinção entre a chave e o estado.

Nas subseções a seguir serão apresentadas mais a fundo o funcionamento do *key schedule* e das funções que compões o AES.

2.1.2.1 Key Schedule

As chaves que serão utilizadas em cada rodada são derivadas pelo *key schedule* a partir da chave utilizada na cifra. O *key schedule* possui duas funções sendo elas a expansão da chave principal e a seleção da chave da rodada. O tamanho total ocupado pelas chaves da rodada é igual ao tamanho do bloco multiplicado pelo número de rodadas mais um. Como exemplo podemos tomar uma encriptação que utiliza AES com chave se 256 bit, $Nr = 14$. Para essa encriptação seriam utilizados 1920 bits para o armazenamento das 15 chaves de rodada necessárias e um bloco de 128 bits. Com o espaço alocado a chave da cifra é expandida e alocada em um vetor subdividido em palavras de 4 bytes. As chaves de cada rodada são selecionadas com a primeira rodada escolhendo as quatro primeiras palavras, totalizando 128 bits, a segunda rodada selecionando as quatro palavras subsequentes e assim por diante [3]. Em casos onde as limitações de memória são grandes as chaves de cada rodada podem ser geradas logo antes de serem utilizadas, diminuindo a necessidade de espaço para armazenamento.

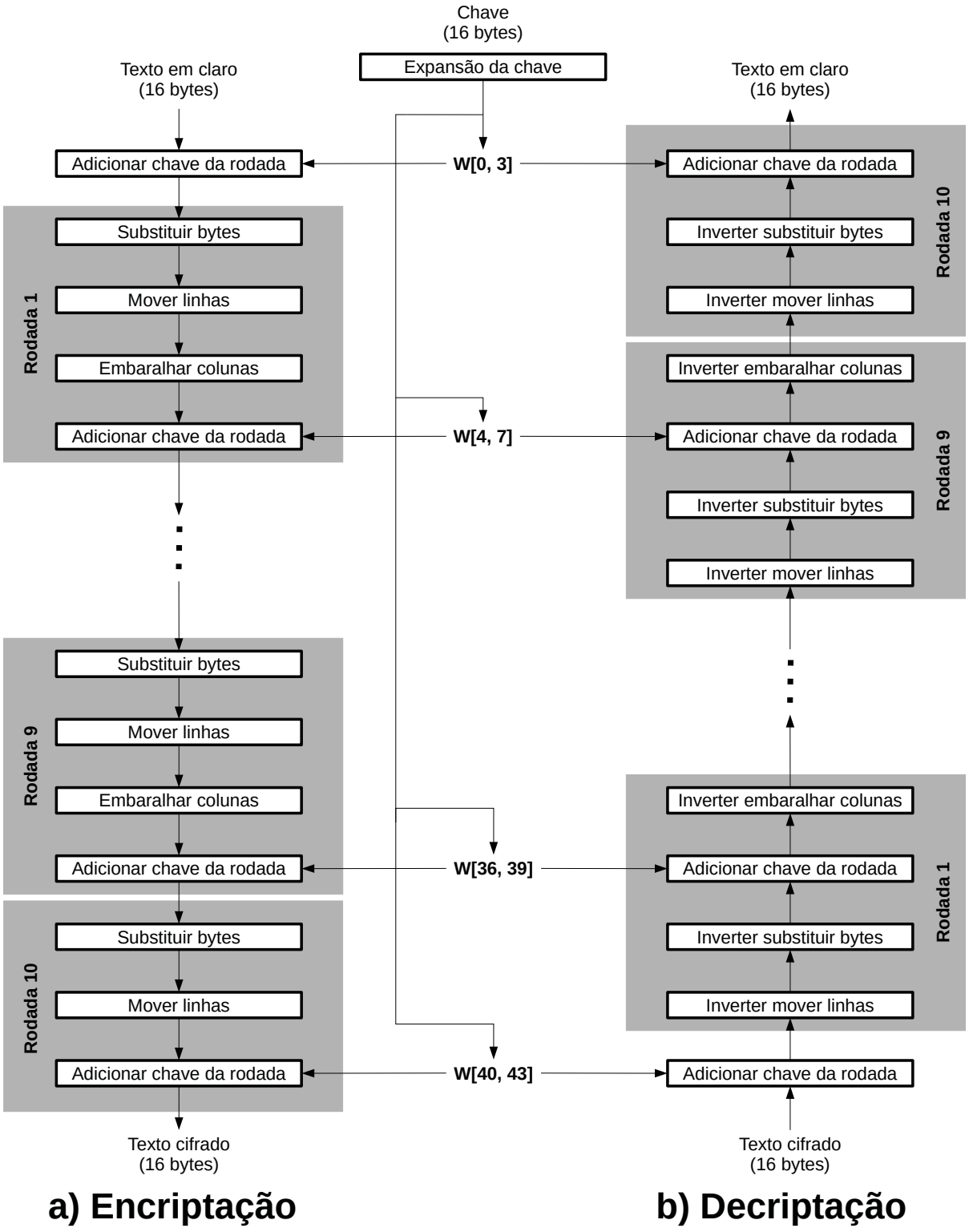


Figura 2.2: Algoritmo AES para encriptação e decriptação, baseado em [2].

2.1.2.2 SubBytes

A primeira função de transformação que será analisada é a chamada *SubBytes*. *SubBytes* é uma substituição não linear de bytes, de onde seu nome é derivado, operando em cada byte do estado de forma independente [3]. A operação ocorre substituindo cada byte do estado por um byte contido na tabela de substituição, chamada de *S-box*. A tabela de substituição é inversível,

característica necessária para a decriptação da mensagem. A tabela de substituição foi criada combinando o inverso multiplicativo de cada um dos bytes, dentro de um corpo finito de tamanho 256, com uma transformação afim inversível de modo a evitar ataques baseados em propriedades algébricas. Outra propriedade utilizada na tabela é que o byte resultante da substituição sempre será diferente do byte de entrada da função, evitando pontos fixos [3]. Como a tabela pode ser pré computada o processo precisa apenas fazer uma consulta para escolher o valor que será substituído. Para cada byte os 4 bits mais significativos indicam a linha da tabela que deve ser escolhida enquanto os 4 bits menos significativos indicam a coluna.Essa busca otimiza o a implementação uma vez que não é necessário recalcular os valores cada vez que se utilizar o algoritmo AES. A Figura 2.3 ilustra o funcionamento da função *SubBytes* enquanto a Tabela 2.1 mostra a tabela *S-box*, conforme definida para o AES e a Tabela 2.2 mostra a tabela utilizada para decriptação durante a operação inversa.

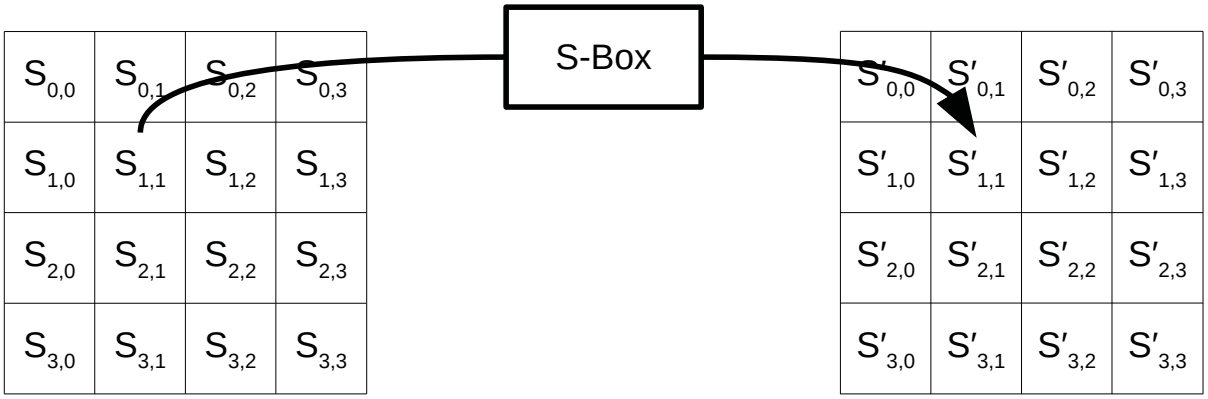


Figura 2.3: Função de transformação *SubBytes*, baseada em [3].

Tabela 2.1: Tabela S-box direta conforme definida para o AES, extraída de [2].

		y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
x	**0**	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Tabela 2.2: Tabela S-box inversa conforme definida para o AES, extraída de [2].

		y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
x	**0**	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

2.1.2.3 ShiftRows

A próxima função de transformação que é utilizada durante as rodadas é a chamada *ShiftRows*. Na função *ShiftRows* cada linha do estado é deslocado circularmente, de modo que um elemento que saia da matriz é adicionado ao outro lado dela. Cada linha é deslocada um número diferente de posições que são proporcionais ao número de sua linha, dessa maneira a linha zero não é deslocada, a linha um é deslocada de um byte, a linha dois de dois bytes e a linha três de três bytes. Os deslocamentos são feitos para a esquerda, em direção aos bytes mais significativos. Para a operação inversa basta realizar a mesma operação somente substituindo o tamanho do deslocamento para três, dois e um para as colunas um, dois e três, respectivamente [3]. A Figura 2.4 ilustra o deslocamento das linhas de uma estado.

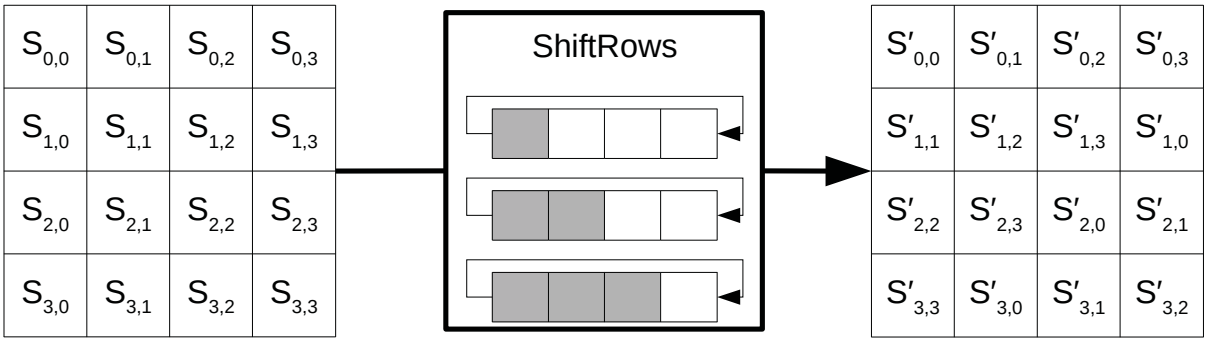


Figura 2.4: Função de transformação *ShiftRows*, baseada em [3].

2.1.2.4 MixColumns

MixColumns é a terceira função de transformação aplicada ao estado em cada uma das rodadas do algoritmo. A função *MixColumns* é a única das funções de transformação que não

é executada durante a última rodada de processamento. Esta função é executada considerando cada uma das colunas de um estado como um polinômio sobre o corpo finito de tamanho 256 e multiplicando-o por um polinômio fixo módulo $x^4 + 1$ [3]. Como a operação pode ser vista de forma matricial é mais simples imaginar a operação apenas como sendo uma multiplicação entre a matriz do estado e uma matriz constante. A Figura 2.5 mostra a relação entre as colunas de entrada e de saída da função. A matriz constante utilizada pela função *MixColumns* pode ser vista na Equação 2.1. Essa matriz relaciona o estado de entrada, com elementos S, com o estado de saída, com elementos $S^{'}$. Embora a Figura 2.5 mostre a operação sobre apenas uma coluna a Equação 2.1 tem como resultado o estado inteiro. Para a decriptação é necessário utilizar a inversa da matriz constante.

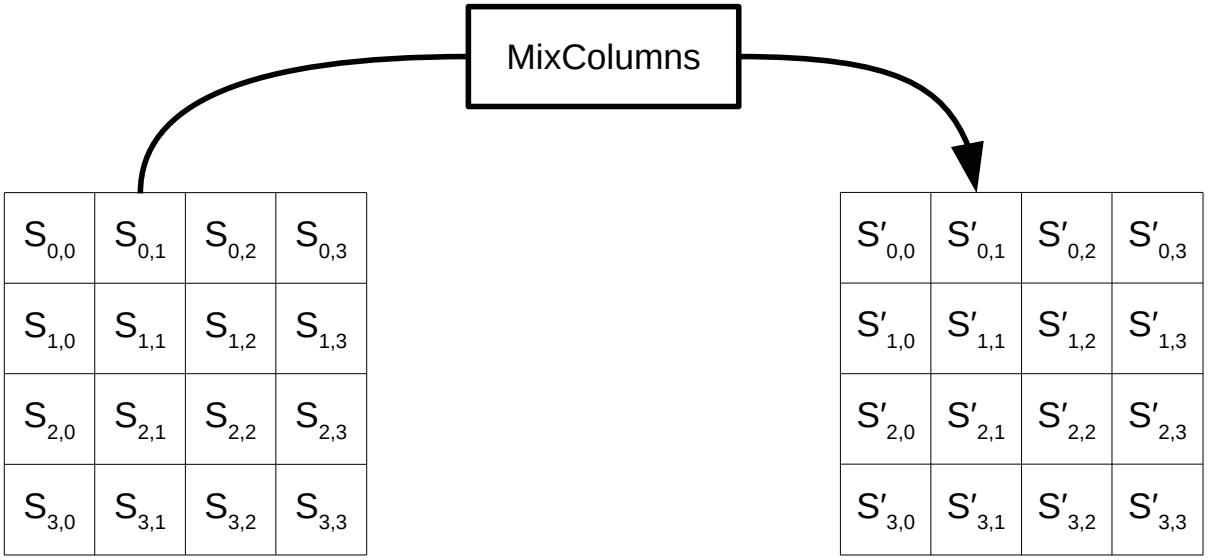


Figura 2.5: Função de transformação *MixColumns*, baseada em [3].

$$\begin{bmatrix} S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\ S'_{1,0} & S'_{1,1} & S'_{1,2} & S'_{1,3} \\ S'_{2,0} & S'_{2,1} & S'_{2,2} & S'_{2,3} \\ S'_{3,0} & S'_{3,1} & S'_{3,2} & S'_{3,3} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix} \quad (2.1)$$

2.1.2.5 AddRoundKey

A função *AddRoundKey* é a função de transformação mais utilizada, sendo usada uma vez antes de se iniciarem as rodadas e mais uma vez em cada uma das rodadas, incluindo a rodada final. É a mais simples das funções, tendo como funcionalidade apenas a realização da operação XOR entre todos os bits do estado e os bits da chave da rodada, que é calculada pelo *key schedule* derivada da chave principal da cifra. Como a aplicação de dois XOR com o mesmo dado retorna ao estado inicial a função *AddRoundKey* é sua própria inversa, que é utilizada durante o processo de decriptação [3]. A Figura 2.6 ilustra o funcionamento da função, sendo W a matriz representando a chave da rodada S o estado de entrada e $S^{'}$ o estado de saída.

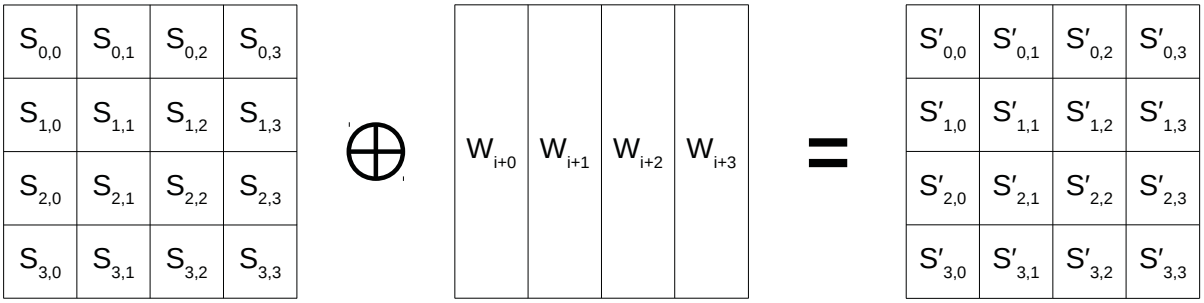


Figura 2.6: Função de transformação *AddRoundKey*, baseada em [3].

2.1.2.6 Possibilidade de otimização

Uma vez definidas as funções de transformação que serão executadas em cada rodada elas podem ser combinadas em um único conjunto de tabelas, permitindo implementações extremamente rápidas em processadores de 32 bits ou mais. Inicialmente as funções *SubBytes* e *ShiftRows* apenas substituem um byte por outro encontrado em uma tabela e trocam a posição do byte na matriz de estado, respectivamente. Essas duas operações podem ser unidas em uma que usa o resultado do *SubBytes* para substituir o a posição de destino do byte após a operação *ShiftRows*. A Equação 2.2 ilustra esse processo em forma matricial, onde a representa um estado. Os índices apresentados nas equações a seguir devem ser considerados módulo quatro. O símbolo *S* representa a função *SubBytes*, os elementos *a* representam os elementos do estado, os elementos *b* representam a saída da função *SubBytes* enquanto *c* representa a saída da função *ShiftRows*.

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j-1} \\ b_{2,j-2} \\ b_{3,j-3} \end{bmatrix} \text{ e } \mathrm{b}_{i,j} = \mathrm{S}\left[a_{i,j}\right] \tag{2.2}$$

Utilizando o resultado da saída da unificação de *SubBytes* e *ShiftRows*, *c*, como entrada é possível representar as operações *MixColumns* e *AddRoundKey* em uma única equação como visto em 2.3 a seguir, onde *k* representa a chave da rodada, *d* representa a saída da função *MixColumns* e *e* a saída da função *AddRoundKey*.

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \text{ e } \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} \tag{2.3}$$

Substituindo os valores de c da Equação 2.3 pelos valores da Equação 2.2, assim como os valores de d, consegue-se chegar à Equação 2.4, que pode ser expressa como uma combinação linear de vetores como ilustrado na Equação 2.5.

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S\left[a_{0,j}\right] \\ S\left[a_{1,j-1}\right] \\ S\left[a_{2,j-2}\right] \\ S\left[a_{3,j-3}\right] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \tag{2.4}$$

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \mathrm{S}\left[a_{0,j}\right] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus S\left[a_{1,j-1}\right] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus S\left[a_{2,j-2}\right] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus S\left[a_{3,j-3}\right] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \tag{2.5}$$

Todos os fatores que multiplicam os vetores podem ser encontrados realizando uma simples busca na tabela *S-box*, desta forma é possível que cada parte da expressão seja computada em novas tabelas, conforme mostram as equações em 2.6.

$$\mathrm{T}_0\left[a\right] = \begin{bmatrix} S\left[a\right] \bullet 02 \\ S\left[a\right] \\ S\left[a\right] \\ S\left[a\right] \bullet 03 \end{bmatrix}, \mathrm{T}_1\left[a\right] = \begin{bmatrix} S\left[a\right] \bullet 03 \\ S\left[a\right] \bullet 02 \\ S\left[a\right] \\ S\left[a\right] \end{bmatrix}, \mathrm{T}_2\left[a\right] = \begin{bmatrix} S\left[a\right] \\ S\left[a\right] \bullet 03 \\ S\left[a\right] \bullet 02 \\ S\left[a\right] \end{bmatrix} e \mathrm{T}_3\left[a\right] = \begin{bmatrix} S\left[a\right] \\ S\left[a\right] \\ S\left[a\right] \bullet 03 \\ S\left[a\right] \bullet 02 \end{bmatrix} \tag{2.6}$$

Substituindo os valores mostrados nas equações 2.6 e substituindo na Equação 2.5 chega-se ao resultado apresentado na Equação 2.7. Desta forma chega-se em uma expressão que mostra que todas as operações podem ser substituídas por apenas 16 buscas em tabela e 16 operações XOR em cada rodada [3], aumentando significativamente a performance do processo.

$$\mathrm{e}_j = T_0\left[a_{0,j}\right] \oplus T_1\left[a_{1,j-1}\right] \oplus T_2\left[a_{2,j-2}\right] \oplus T_3\left[a_{3,j-3}\right] \oplus k_j \tag{2.7}$$

Cada uma das tabelas é composta por 256 entradas de 4 bytes cada, totalizando 4KiB de espaço para armazenamento das mesmas. É possível notar pelas expressões das equações apresentadas em 2.6 que cada uma das tabelas é apenas a rotação de uma das outras, dessa forma em ambientes em que a falta de espaço é um fator limitante é possível armazenar apenas umas das tabelas e computar as demais. Na rodada final não é feita a operação *MixColumns*, o que significa que a tabela *S-box* deve ser utilizada em vez da tabela T, porém isso não é um problema pois a tabela *S-box* pode ser recuperada através da tabela T. Outro ponto importante em relação à expressão encontrada é o fato de que como ela é baseada em buscas em quatro tabelas separadas essas buscas podem ser paralelizadas, aumentando ainda mais a performance do processo.

2.1.3 Modos de operação

O AES, assim como os algoritmos de cifras de bloco de uma maneira geral, não é capaz de cifrar mensagens maiores que 128 bits, tamanho do seu bloco, de maneira direta. Esta

limitação levou a criação de diversos modos de operação que são utilizados para que torne viável a utilização dos algoritmos de maneira geral. Muitos desses modos datam de muito antes da criação do AES e cada um deles possui características distintas, possuindo vantagens e desvantagens em relação aos demais modos.

Diferentes métodos podem gerar propriedades diferentes na mensagem. Enquanto alguns modos trazem apenas confidencialidade à mensagem, impedindo que pessoas não autorizadas tenham acessos, outros geram apenas autenticidade, permitindo a correta identificação do criador da mensagem, enquanto alguns buscam fazer uma encriptação autenticada, unindo fortemente a autenticação com a cifragem da mensagem [19].

Nas subseções a seguir serão apresentados os modos *Electronic Codebook* (ECB), *Cipher Block Chaining* (CBC), *Cipher Feedback* (CFB), *Output Feedback* (OFB) e *Counter* (CTR) que são modos responsáveis por trazer confidencialidade à mensagem que se deseja criptografar.

2.1.3.1 ECB

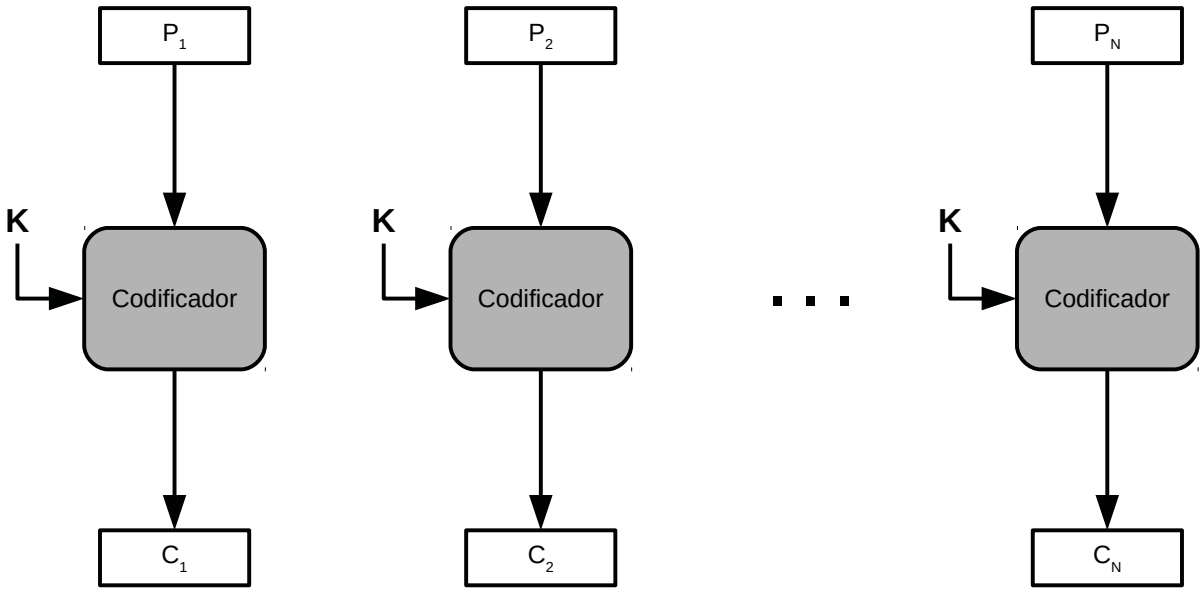
O modo ECB é um modo de criptografia no qual, para uma dada chave, um bloco fixo do texto em claro é convertido em um bloco de criptografado, possuindo comportamento similar à um livro de códigos em que cada palavra é trocada por uma outra palavra correspondente [20]. O modo ECB é um dos quatro modos clássicos e foi criado para o uso com o DES a mais de 40 anos atrás [19]. No modo ECB é utilizado o algoritmo de encriptação direto quando se deseja encriptar e seu inverso quando se deseja decriptar. A encriptação e decriptação utilizando o modo ECB pode ser feita completamente em paralelo, uma vez que cada bloco passa por um processo de encriptação que não depende dos demais [20]. Por utilizar blocos inteiros em sua encriptação é necessário que seja realizado o preenchimento do último bloco caso este não possua tamanho necessário, técnica essa chamada de *padding*. A Figura 2.8 ilustra o funcionamento do modo, sendo (a) a encriptação e (b) a decriptação.

Apesar de simples e ter a capacidade de ser paralelizável o método possui alguns problemas bem conhecidos [19]. O primeiro, e possivelmente o mais simples de ser identificado, é o fato de que blocos idênticos do texto em claro levam a blocos idênticos na cifra, que diminui de forma significativa a confidencialidade da cifra. Essa característica permite a observação de padrões na cifra que refletem padrões existentes no texto em claro original. A Figura 2.8 ilustra bem essa situação, onde pode-se ver claramente um pinguim na imagem cifrada, mesmo sem se ter os detalhes de cada parte do original.

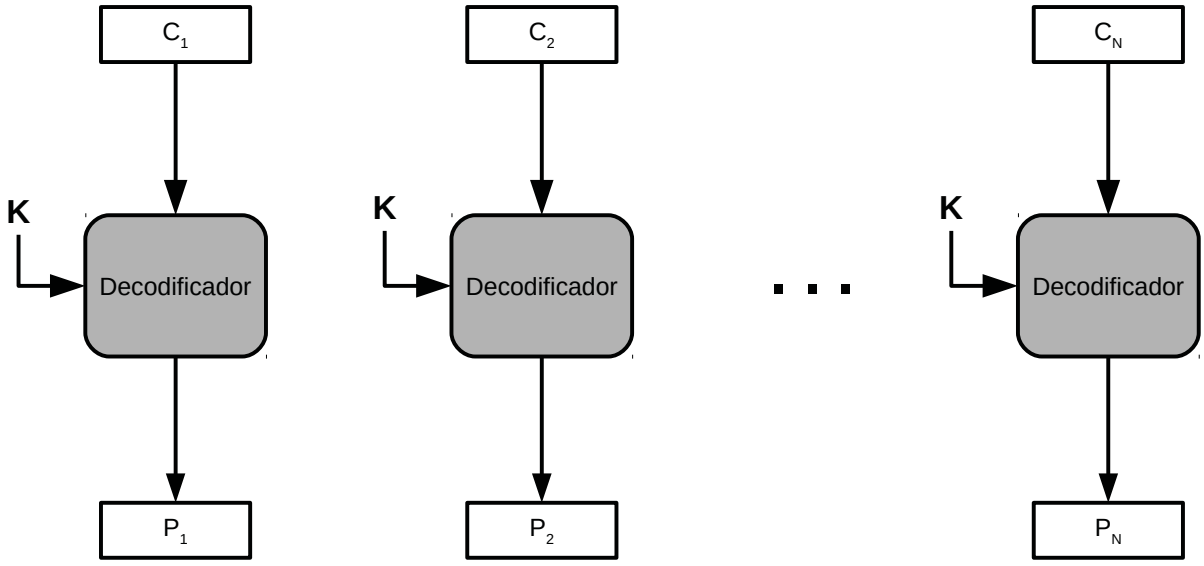
De maneira geral o modo ECB nunca foi tratado como um modo interessante por criptógrafos. O modo possui falhas que podem comprometer a confidencialidade da informação, trazendo segurança apenas para informações que possuam característica realmente aleatória, livre de padrões. Este modo não deve ser considerado ou utilizado como modo principal para trazer confidencialidade aos dados, se mantendo presente nos padrões não por sua utilizada direta mas sim por sua capacidade de ser parte importante na construção de outros modos melhores [19], como por exemplo o modo CTR que será apresentado na Subseção 2.1.3.5.

2.1.3.2 CBC

O modo CBC é um dos modos de operação criados para ser utilizado com o cifrador DES e nele é feita uma ligação entre o bloco de texto em claro que está sendo criptografado com o bloco cifrado anterior. Esta característica faz com que mesmo blocos iguais sejam cifrados para cifras diferentes, impedindo que se encontrem padrões do texto em claro analisando o texto cifrado, como ocorre no modo ECB. O modo CBC necessita de um vetor de inicialização (IV)



a) Encriptação



b) Decriptação

Figura 2.7: Funcionamento do modo ECB, baseado em [2].

para ser combinado com o primeiro texto em claro, pois não existe texto cifrado anterior. A combinação entre o bloco de texto em claro e o bloco cifrado anterior, ou IV no primeiro bloco, é feito realizando um XOR entre os dois blocos. O IV não necessita ser mantido em segredo, diferente da chave de encriptação, porém deve ser imprevisível para que a cifra seja segura [20]. Por se utilizar de blocos completos para a criptografia o modo CBC precisa realizar *padding* quando necessário, assim como no modo ECB.

Para se fazer a decriptação de um bloco que foi encriptado com o modo CBC é necessário fazer o processo reverso ao de encriptação do bloco e realizar o XOR com o bloco criptografado anterior, ou com o IV no caso do primeiro bloco. Por possuir essas características de utilização do

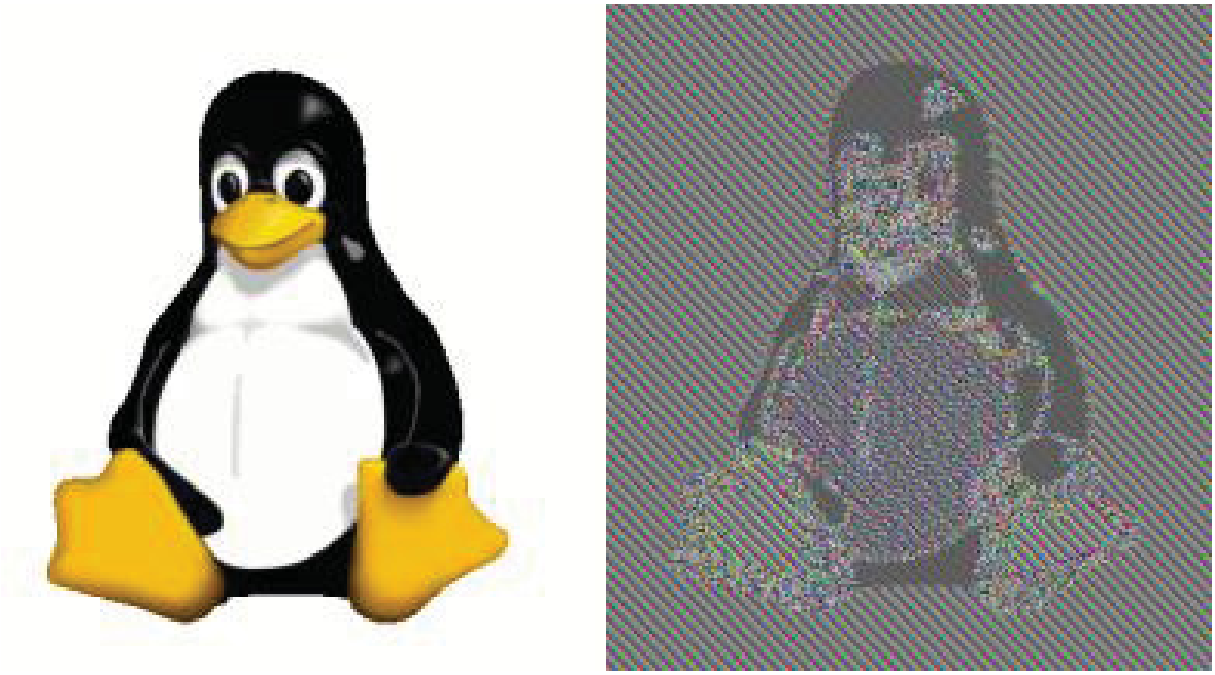


Figura 2.8: Imagem antes e depois de ser encriptada utilizando modo ECB [4].

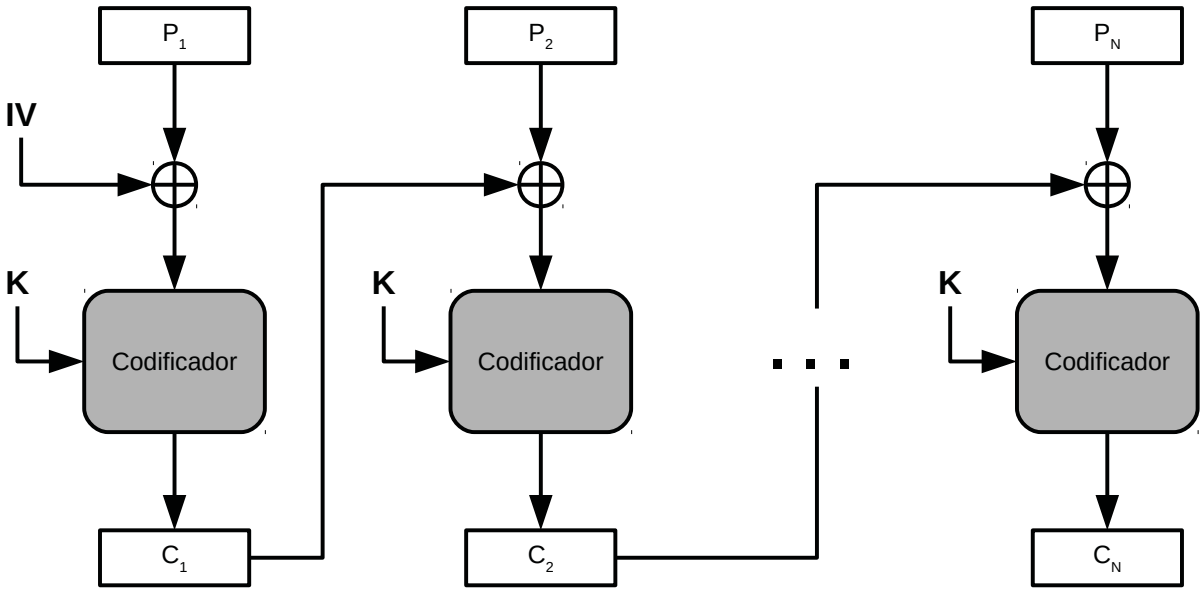
bloco cifrado anterior o modo CBC precisa ser feito de forma linear durante a encriptação, porém pode ser feito de forma paralela durante a decriptação, uma vez que todos os blocos cifrados já estão disponíveis para serem utilizados [20]. A Figura 2.9 ilustra o funcionamento do modo.

De maneira geral o modo CBC é considerado seguro contanto que alguns cuidados sejam tomados na escolha do IV. O IV utilizado deve ser imprevisível. Com a escolha aleatória do IV a cifra gerada torna-se praticamente indistinguível de um bloco aleatório [19]. A utilização de um *nonce*, valor que só é utilizado uma vez mas que não necessita ser aleatório, como IV torna o modo vulnerável e sua utilização não é recomendada. É bastante comum encontrar na literatura que a geração de um IV pode ser feita a partir de um *nonce* criptografado com a mesma chave utilizada no processo de criptografia do texto em claro [2][21][22][20], porém como mostrado por no libro *Introduction to Modern Cryptography*[19] esta abordagem possui os mesmos problemas de se utilizar diretamente um *nonce* sem que este seja imprevisível. Uma solução simples apresentada é a utilização de outra chave para a criptografia do *nonce*, tornando-o imprevisível em relação à cifra.

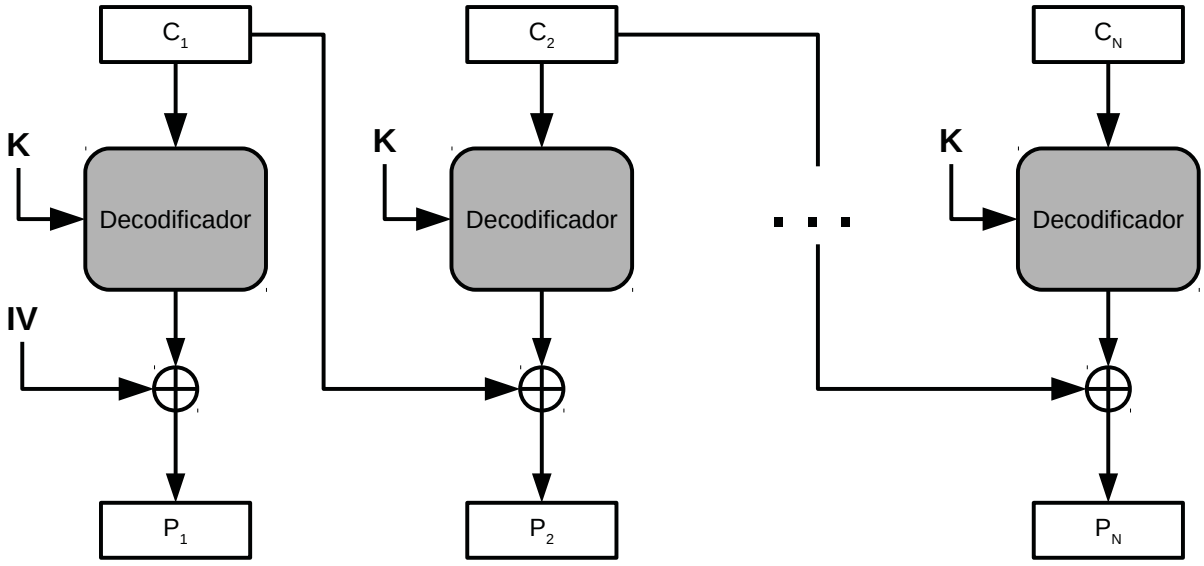
2.1.3.3 CFB

O processamento no modo CFB consiste em utilizar a saída do segmento criptografado anterior como entrada do próximo segmento, utilizando um IV para a encriptação do primeiro segmento, e realizar um XOR dessas saídas com o texto em claro para gerar o texto cifrado. Para se fazer a decriptação o processo é o mesmo e, no final, é feito o XOR das saídas com o texto cifrado, revelando o texto em claro. O IV, assim como no modo CBC, deve ser criado de forma imprevisível, não podendo ser utilizado apenas um *nonce*.

O modo CFB necessita de um parâmetro adicional chamado de s que deve ser maior que zero e menor ou igual ao tamanho total de um bloco, denominado b. Este parâmetro geralmente vem incorporado ao nome do modo para que seja possível saber qual está sendo usado, 8-bit CFB por exemplo. Para cada bloco que é processado s bits do texto em claro são transformados em s bits da cifra. Para o primeiro bloco é utilizado um IV de tamanho b que gera um saída cifrada de



a) Encriptação

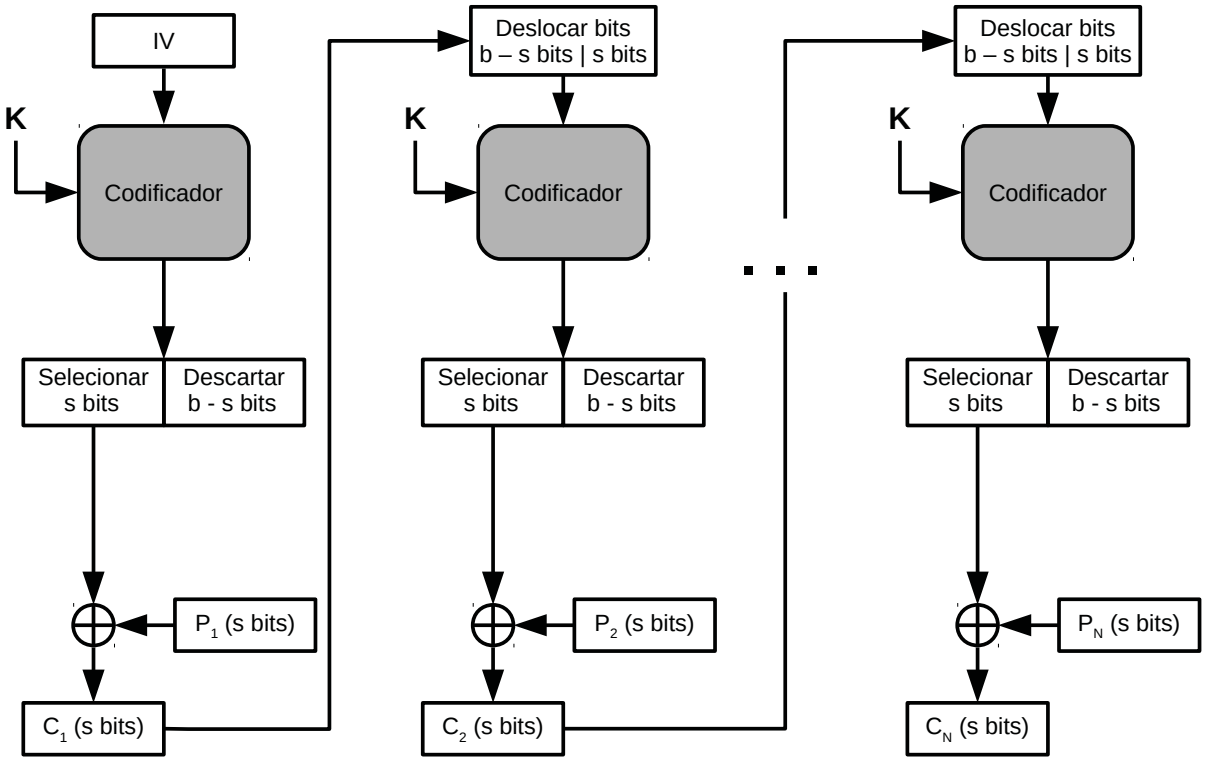


b) Decriptação

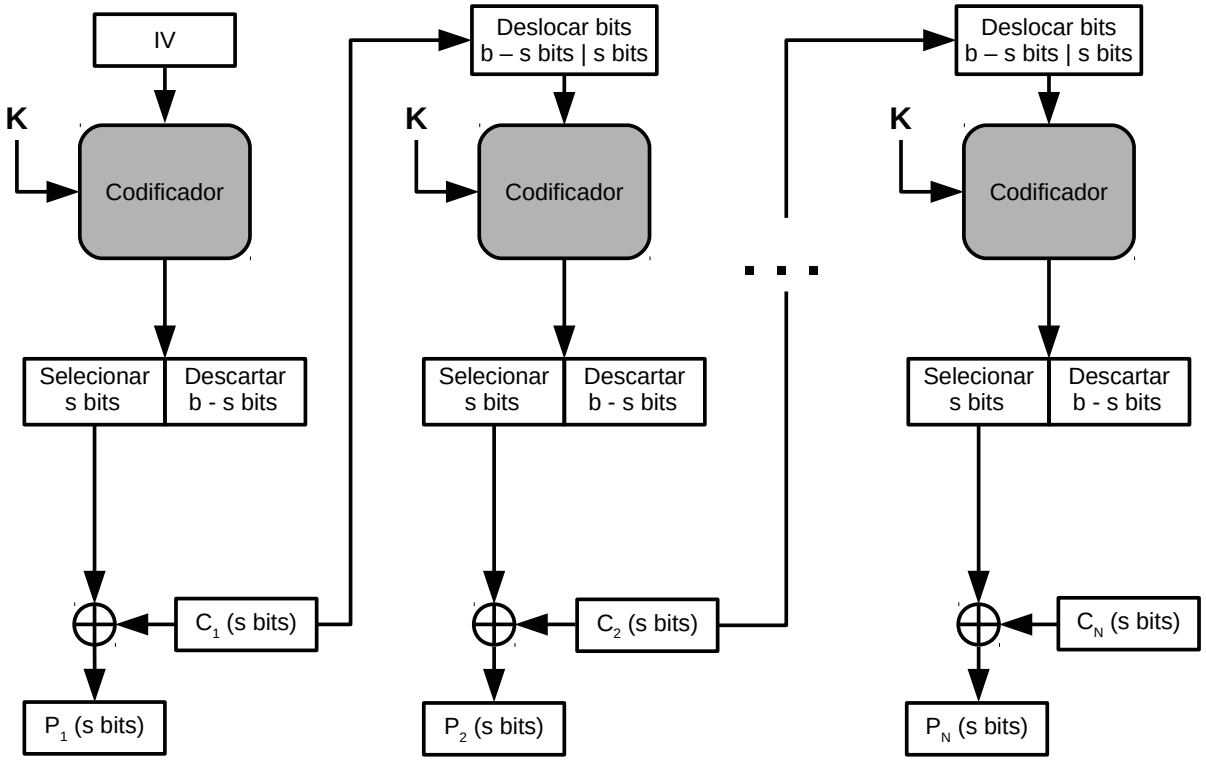
Figura 2.9: Funcionamento do modo CBC, baseado em [2].

tamanho também b. Os s bits mais significativos do bloco de saída são então utilizados para fazer um XOR com os s primeiros bits do texto em claro, gerando os s primeiros bits do texto cifrado. Para o próximo passo é necessário gerar a nova entrada para o algoritmo. Para a geração do novo bloco de entrada são concatenados os $b - s$ bits menos significativos da entrada utilizada na etapa anterior com os s bits de texto cifrado gerados na etapa anterior. Com a nova entrada o processo se repete até que todos os bits do texto em claro sejam cifrados [20][19]. A Figura 2.10 esclarece o funcionamento do modo CFB.

Devido ao modo como é feita a encriptação do texto em claro é possível ver que não é necessário que o texto possua tamanho múltiplo de s e, com isso, não é necessária a utilização de



a) Encriptação



a) Decriptação

Figura 2.10: Funcionamento do modo CFB, baseado em [2].

padding. Dessa forma pode-se concluir que o modo CFB é uma cifra de fluxo, diferente dos modo ECB e CBC que eram cifras de bloco. Apesar dessa diferença a escolha do IV utilizado pelo modo CFB deve tomar os mesmo cuidados que no modo CBC, não podendo ser previsível e não devendo ser gerado a partir da criptografia de um *nonce* a partir da mesma chave.

O modo CFB, devido a sua natureza, em que a entrada do bloco seguinte depende tanto da entrada como da saída anterior, não pode ser paralelizado em sua encriptação, porém pode ser na decriptação, embora seja necessário primeiro gerar todos os blocos de entrada. Essa característica traz, no entanto, a propriedade de sincronização, indicando que um erro em uma parte da cifra não afeta a cifra inteira, ficando contida apenas naquela parte e não prejudicando a decriptação do resto da mensagem [19]. Esta propriedade pode ser desejada em algumas situações,o que colocaria o modo CFB em vantagem sobre os demais.

2.1.3.4 OFB

O modo OFB funciona gerando uma sequencia de blocos cifrados a partir de um IV. Inicialmente o IV é utilizado como entrada no cifrador, gerando um bloco cifrado que é utilizado como entrada. Em cada novo passo é utilizado a saída do passo anterior como entrada do passo seguinte. O processo continua até que seja gerada uma quantidade de bits maior ou igual ao tamanho do texto em claro. Para a geração do texto cifrado é realizada a operação de XOR entre cada um dos blocos de saída com os blocos do texto em claro. O processo é exatamente o mesmo para se fazer a decifragem, a única ressalva é que o XOR é feito com o texto cifrado para se gerar o texto em claro [20][19]. A Figura 2.11 mostra o funcionamento do modo OFB.

Para a inicialização da cifragem é utilizado um IV da mesma forma que é utilizado nos modos CBC e CFB, porém, diferente desses modos o IV para o modo OFB deve ser um *nonce*, não podendo ser repetido um mesmo IV com uma mesma chave. A necessidade de ser um *nonce*, no entando, não é a única restrição que se deve aplicar durante a escolha do IV, devendo se verificar a não previsibilidade do mesmo e não devendo se utilizar a cifragem de um *nonce* utilizando a mesma chave, assim como nos modos CBC e CFB.

Na encriptação utilizando OFB é feito o XOR dos blocos de saída com o texto em claro. Caso o texto em claro não possua bits para completar o último bloco de saída, então os últimos bits não são utilizados, sendo descartados. Dessa maneira o modo de operação OFB é visto como um método de cifragem de fluxo, assim como o CFB.

Como, tanto na encriptação quanto na decriptação uma parte do processo depende do resultado da parte anterior não é possível paralelizar o processamento do modo OFB. De modo similar, como os bloco utilizados para gerar o texto cifrado não dependem do texto em claro, sendo apenas operado junto com ele no final, é possível pré computá-los, acelerando o processo de encriptação ou decriptação.

Como mostrado no livro *Introduction to Modern Cryptography*[19] a segurança do modo OFB pode ser provada uma vez que a segurança do modo CBC foi provada. É possível considerar o modo OFB como uma cifra no modo CBC de um texto em claro de tamanho zero. Desta forma a prova se torna direta e mostra que, considerando a escolha correta de IV, a segurança do modo CBC e OFB são equivalentes.

2.1.3.5 CTR

Comparado com os quatro modos mais antigos que foram criados inicialmente para funcionarem junto com o DES, ECB, CBC, CFB e OFB, o modo CTR é mais novo e moderno, sendo criado ignorando algumas das considerações que não fazem mais sentido atualmente. Sua definição é a mais simples de todos os modos, sendo mais facilmente compreendida quando

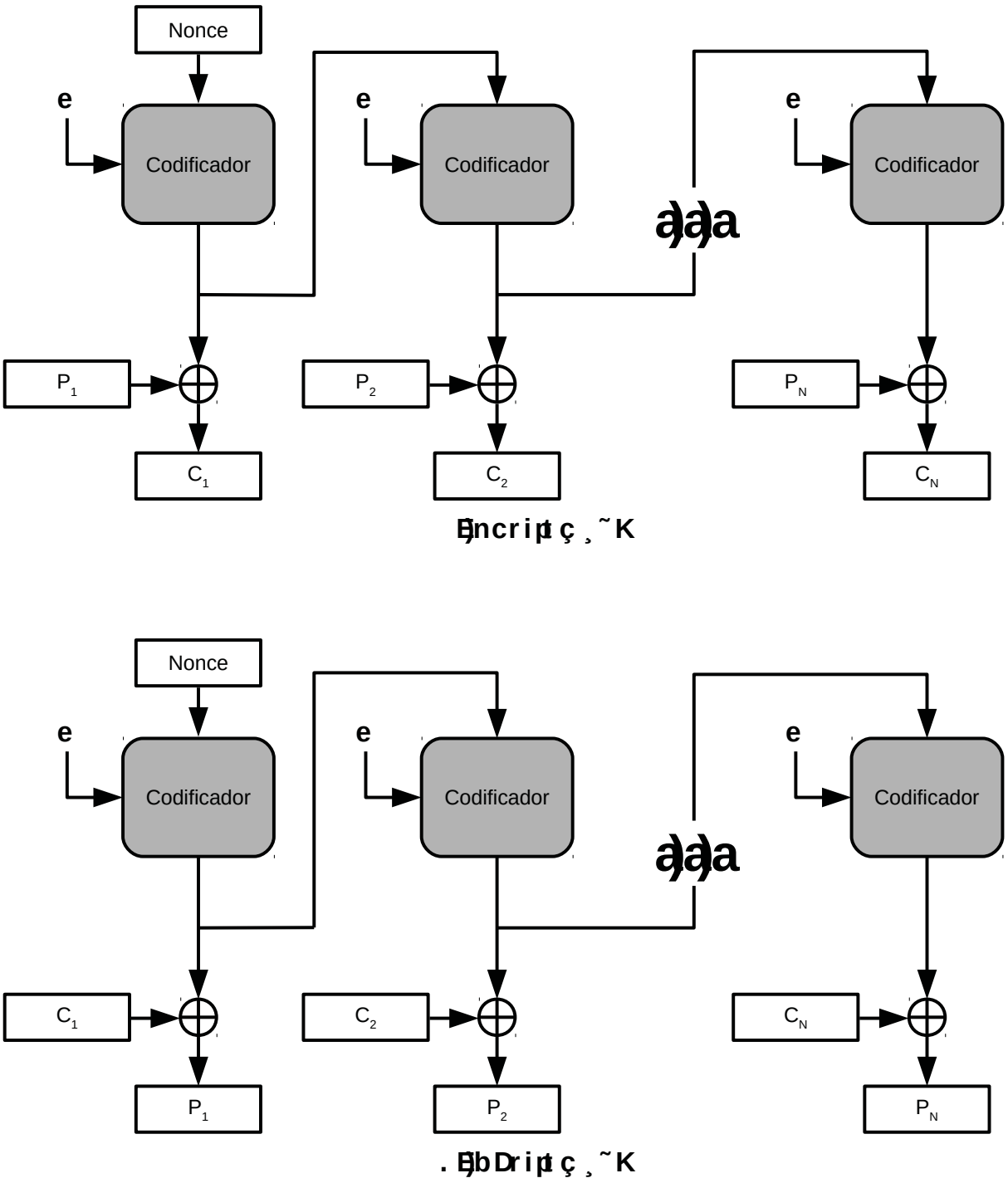


Figura 2.11: Funcionamento do modo OFB, baseado em [2].

comparado com os demais modos. O modo CTR consiste da aplicação de uma cifra de blocos direta em um conjunto de blocos de entrada chamados de contador [20]. A aplicação da cifra nos blocos de entrada gera um conjunto de blocos de saída que, posteriormente, são utilizado em um XOR junto ao texto em claro para gerar o texto cifrado. Da mesma forma como o modo OFB a decriptação do modo CTR é feita da mesma maneira que a encriptação, alterando apenas o texto em claro para o texto cifrado na operação final de XOR. O contador que será utilizado como bloco de entrada deve possuir a propriedade de ser único e nunca deve ser utilizado mais de uma vez com a mesma chave. Diferente do que ocorre em outros modos isso não se restringe

a mensagem, e sim abrange todos os blocos de todas as mensagens que serão cifrados com a mesma chave. O par chave e contador só deve ser utilizado uma vez [20]. A Figura 2.12 mostra o funcionamento do modo CTR.

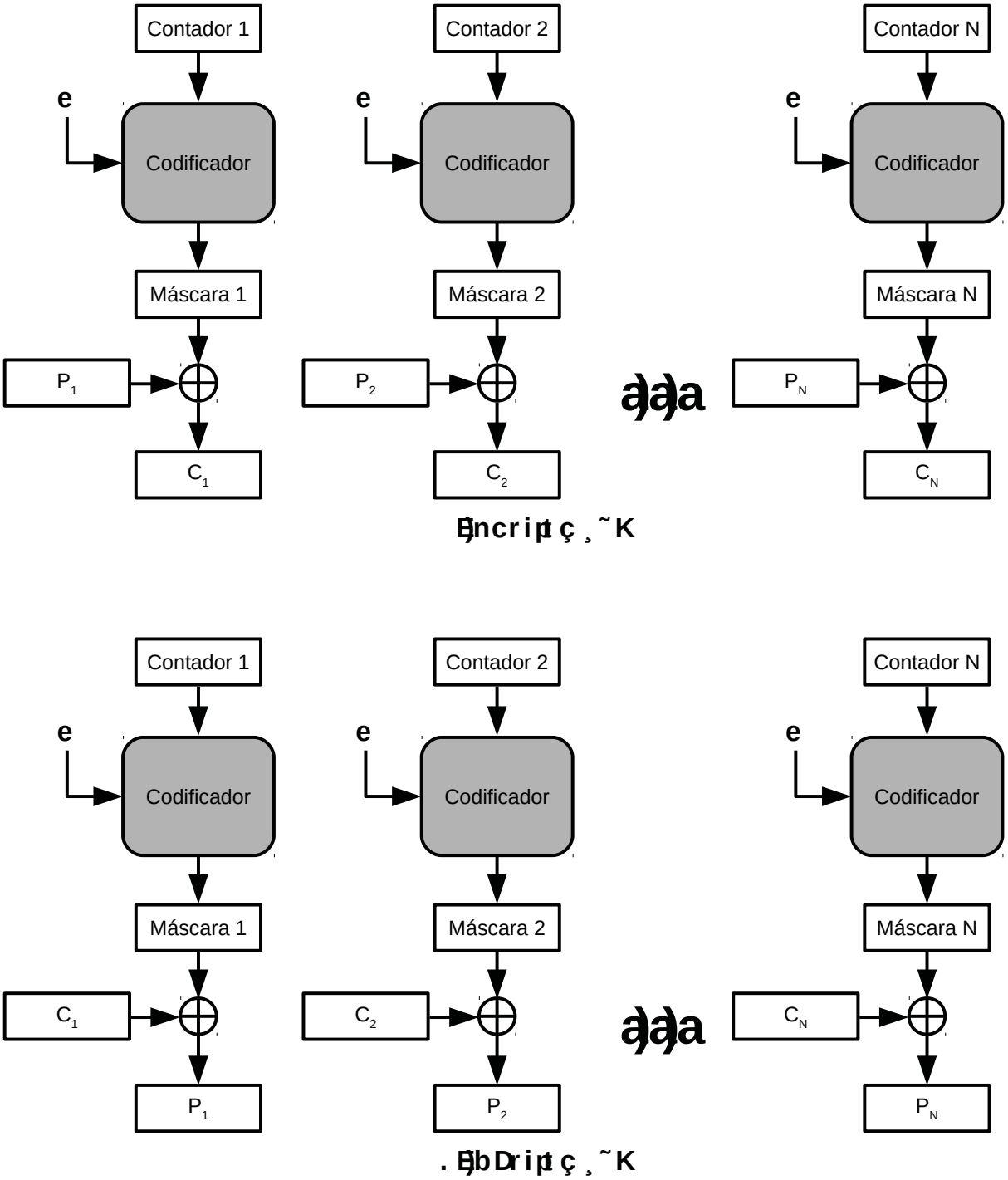


Figura 2.12: Funcionamento do modo CTR, baseado em [2].

Para a encriptação utilizando o modo CTR serão gerados um número de blocos cujo tamanho seja maior ou igual ao da mensagem em claro. Caso o tamanho da mensagem não seja suficiente para completar o último bloco os bits finais serão descartados, não havendo a necessidade de se utilizar *padding*. Por essa característica o modo CTR é definido como modo de cifra de fluxo.

Devido à forma como o modo é definido é possível ver que nenhuma de suas partes depende que nada seja computado anteriormente, dessa maneira o modo CTR pode ser completamente paralelizável. Diferente dos métodos apresentados anteriormente o modo CTR pode ser paralelizado tanto na encriptação quanto na decriptação. Essa característica permite que tanto hardware quanto software possam ser usados de maneira eficiente e rápida durante seu processamento.

Outra característica importante é sua capacidade de pré-processamento. Uma vez que a cifra final é feita a partir do XOR entre o texto em claro e a saída de uma cifra de blocos e as suas entradas não dependem em nenhum ponto do texto em claro é possível pré computar essas saídas. Assim como no modo OFB é possível gerar todas as saídas das funções de maneira prévia e, quando for necessário encriptar ou decriptar algo basta fazer a operação de XOR porém, diferente do OFB o modo CTR também é paralelizável, o que se traduz em ganhos muito maiores de velocidade.

O modo CTR também possui a propriedade de poder realizar acessos aleatórios aos dados cifrados. Quando se deseja acessar alguma informação que se encontra em um bloco do texto cifrado é necessário apenas processar a decifragem daquele bloco em particular. Em outros modos, como o CBC, é necessário computar todos os blocos anteriores para se acessar a informação [19]. No modo OFB só é necessário decriptar o bloco desejado, porém para isso é necessário realizar um volume de processamento equivalente ao de decriptar todos os blocos anteriores.

De um modo geral se acredita que para conseguir características como essa é preciso abrir mão da segurança, porém esse não é o caso. No livro *Introduction to Modern Cryptography*[19] é mostrado que é possível provar que a segurança do modo CTR não é pior que a do modo CBC, quando utilizado de maneira correta. A segurança do modo CTR é bem estudada e bem entendida, o que torna o modo confiável.

Sendo o mais novo e moderno modo de se criptografar e decriptografar dados o modo CTR possui um conjunto de vantagens que nenhum outro modo possui, alguns modos possuem algumas dessas vantagens mas nenhum possui todo o conjunto. Essas características levaram muitos pesquisadores a dizer que não existe mais motivos para se utilizar os modos mais antigos, que devem ser mantidos como padrões apenas por ainda serem utilizados em larga escala em equipamentos mais antigos [19].

2.2 ESTRUTURA DO LINUX E SISTEMAS DE ARQUIVO CRIPTOGRAFADOS

A busca por mais segurança da informação unido ao constante aumento da capacidade computacional dos dispositivos tem levado a um crescimento na busca por métodos de criptografia de dados. Nesta tarefa sistemas de arquivos criptografados são bastante desejados uma vez que trazem confidencialidade a todos os arquivos e diretórios contidos nele, trazendo praticidade e simplicidade na hora de manter os dados seguros.

Para se entender melhor o funcionamento dos sistemas de arquivos criptografados nos ambientes Linux é preciso inicialmente possuir um conhecimento básico sobre a estrutura do sistema operacional, entendendo melhor como são divididos os ambientes e como cada um deles possui características e permissões diferentes, os modos como é possível se gerar sistemas de arquivos e como realizar a criptografia sobre os mesmos.

Nas seções a seguir serão explicados os conceitos básicos necessários para o entendimento de como um sistema de arquivo criptografado em espaço de usuário pode ser implementado. Na Seção 2.2.1 será apresentado um pouco sobre a estrutura e funcionamento do sistema operacional Linux, explorando as possibilidades oferecidas por ele. A Seção 2.2.2 explora um pouco como

é dividido o *kernel* e como são utilizadas suas partes. A Seção 2.2.3 apresenta o subsistema VFS e sua relação com diferentes sistemas de arquivo. Na Seção 2.2.4 será mostrada uma forma de se criar sistemas de arquivos em espaço de usuário no Linux e na Seção 2.2.5 será apresentado o EncFS, biblioteca criada para gerar sistemas de arquivos criptografados no Linux e base da EncFS++. Por fim a Seção 2.2.6 apresentará alguns sistemas de arquivos criptografados implementados em espaço de *kernel*, mostrando outra maneira como podem ser criados.

2.2.1 Estrutura do Linux

De maneira geral é difícil definir com precisão o que é um sistema operacional. Embora não exista dúvida quando se fala que o Linux é um sistema operacional, grande parte dos autores possuem definições diferentes sobre o que significa ser um sistema operacional. Independente da definição apresentada, um sistema operacional sempre exercerá duas funções que o definem como sendo um sistema operacional que são oferecer aos programadores um conjunto abstrato de interfaces simples, em vez das complicadas interfaces diretas para o hardware, e realizar o gerenciamento dos recursos de hardware [5].

O sistema operacional Linux deve, como mostrado no parágrafo anterior, possuir um conjunto de interfaces simples para apresentar aos programadores de aplicações e fazer o gerenciamento do hardware. Para completar essas tarefas o Linux é dividido em diversas camadas de abstração que vão desde o mais complexo e difícil de entender, ao nível de hardware, até o nível mais simples, ao nível de usuário. A Figura 2.13 mostra de forma abstrata a ligação entre as camadas.

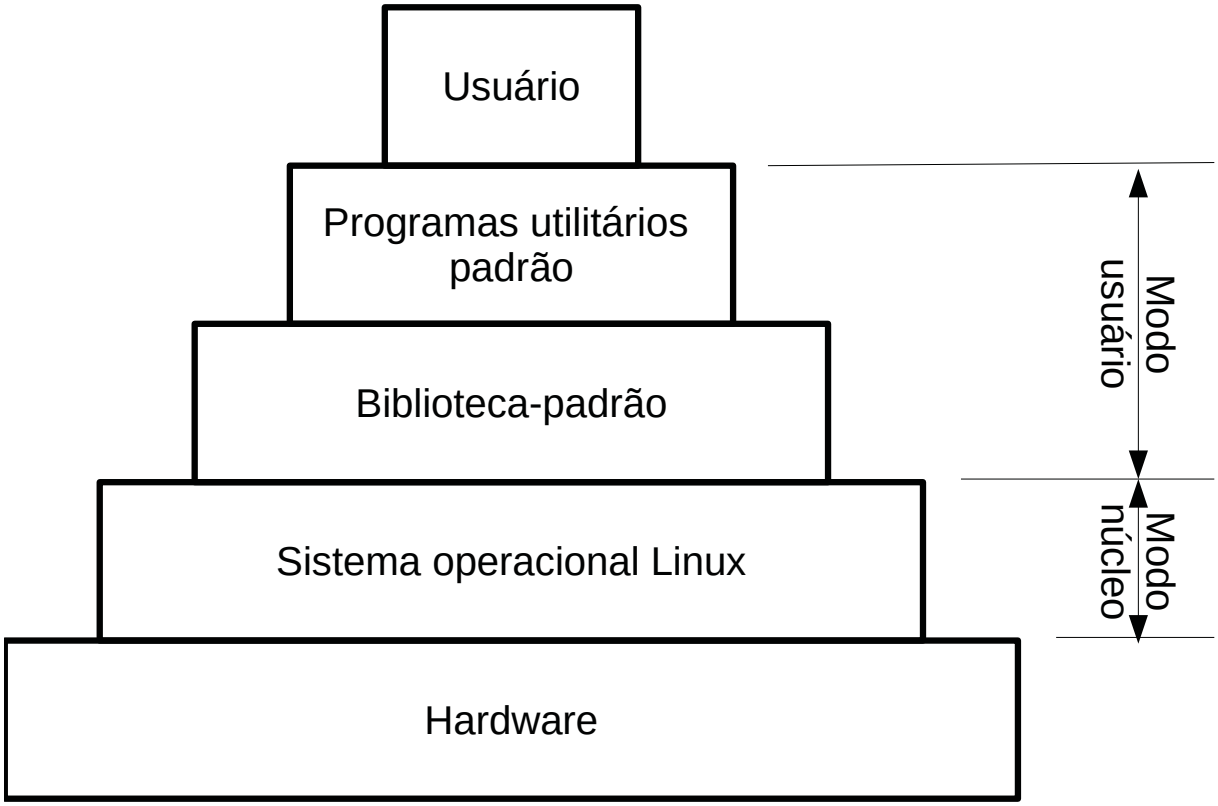


Figura 2.13: Modelos de camadas de um sistema operacional Linux, baseado em [5].

Como mostrado na Figura 2.13 existem algumas camadas de abstração entre o usuário e o *hardware*. Essas camadas possuem interfaces entre elas de modo que, para os usuários e desenvolvedores de aplicação as chamadas complexas de *hardware* sejam vistas como chamadas

mais simples e padronizadas. No livro *Modern Operating Systems*[5] essa visão é chamada de sistema operacional como máquina estendida.

Também é possível ver na Figura 2.13 a existência de dois modos de operação, chamados modo usuário e modo núcleo, mais comumente chamado de modo *kernel*. Modo usuário é um modo não privilegiado responsável pela execução de aplicações do usuário, como editores de texto e navegadores de internet por exemplo. Neste modo são gerenciados os recursos encontrados no chamado espaço de usuário que corresponde a um subconjunto dos recursos disponibilizados pela máquina [23] e pode realizar algumas chamadas de sistema, não conseguindo acesso direto ao *hardware* e nem acesso fora do espaço alocado pelo *kernel*. O modo *kernel*, por outro lado, é um modo privilegiado que possui acesso a todo o *hardware*. Quando um processo realiza uma chamada de sistema, requisitando algum recurso, o *hardware* muda de modo usuário para modo *kernel* para que possa ser realizado o acesso e disponibilização desses recursos para processo requisitante [5][24]. A utilização dos dois modos é necessária uma vez que processos de usuário executados em modo *kernel* apresentam grades riscos para o sistema, podendo até mesmo apagar o sistema operacional caso realizem escrita em uma área errada da memória. Os modos também permitem a sincronização de chamadas de *hardware*, impedindo que dois processos de usuário realizem um mesma requisição de forma simultânea e gerando resultados incorretos. A Seção 2.2.2 falará um pouco mais sobre o que é e como funciona o *kernel* do sistema operacional Linux.

2.2.2 Kernel

O *kernel* é o núcleo do sistema operacional Linux, sendo a parte mais interna e próxima do *hardware*, sendo o extremo oposto da interface com o usuário, normalmente chamada de *shell* [5] e que é a parte mais externa do sistema e mais longe do *hardware* [23]. O *kernel* é o software responsável por prover serviços básicos para todas as demais partes do sistema, gerenciar o *hardware* e distribuir os recursos do sistemas [23].

Embora o foco das pesquisas acadêmicas seja voltado para os chamados *microkernels* [24], que são aqueles que possuem poucas funcionalidades e apresentam poucas funções para as camadas superiores, os *kernels* do Linux são, de maneira geral, monolíticos, onde cada camada do *kernel* é integrado e são executadas em modo *kernel* [24][5][23]. Diferente dos programas de usuário os *kernels* são, de maneira geral enormes, complexos e possuem vida longa [5]. A Figura 2.14 mostra a evolução do número de linhas de código do sistema operacional Linux ao longo dos anos.

Devido ao pequeno número de funções executadas por *microkernels* estes precisam de passagens explicitas de mensagens entre cada uma de suas camadas o que, de maneira geral, os tornam mais lentos que os *kernels* monolíticos, porém estes possuem vantagens teóricas, como a necessidade de modularização, a facilidade na transição para outro *hardware* e melhor uso da memória RAM [24]. O *kernel* do Linux, apesar de monolítico se utiliza largamente das vantagens encontradas nos *microkernels* sem sofrer com suas perdas de desempenho.

Uma das vantagens encontradas nos *microkernels* e que é aproveitada pelo *kernel* do Linux é a capacidade de se adaptar facilmente para ser utilizado com diferentes *hardwares*. O *kernel* do Linux utiliza o conceito de módulos que são blocos de código que o *kernel* pode carregar enquanto o sistema está sendo executado [5]. Os módulos permitem, entre outras coisas, que novos *hardwares* sejam plugados ao sistema sem a necessidade de parada ou alteração do código do *kernel*.

A utilização de diferentes módulos em conjunto permite que diversas aplicações de usuário funcionem de maneira transparente, uma vez que as chamadas de sistema padrão do *kernel* podem ser traduzidas para os mais variados tipos de *hardware* independente da plataforma em que se encontrem [24]. Além do conceito de módulo também existe o conceito de subsistemas

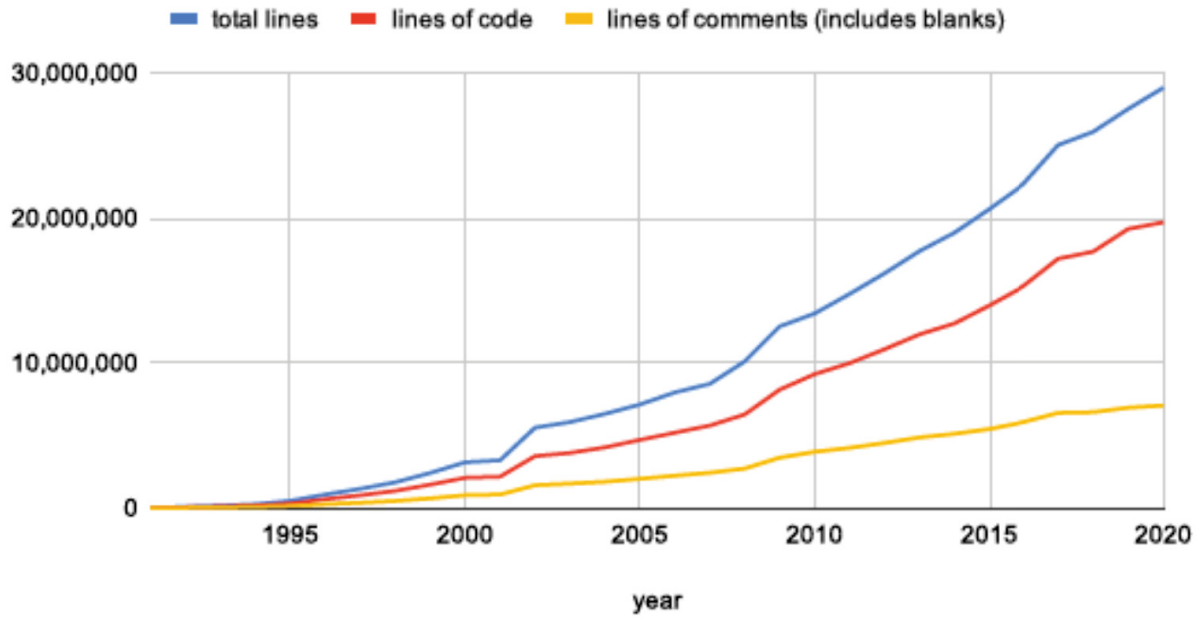


Figura 2.14: Crescimento do número de linhas de código do *kernel* do Linux ao longo dos anos [6].

que são compostos de um ou mais módulos e são responsáveis por implementar alguma função do sistema operacional. Como mostra a Figura 2.15 o *kernel* é composto por módulos e subsistemas, além de sua parte monolítica.

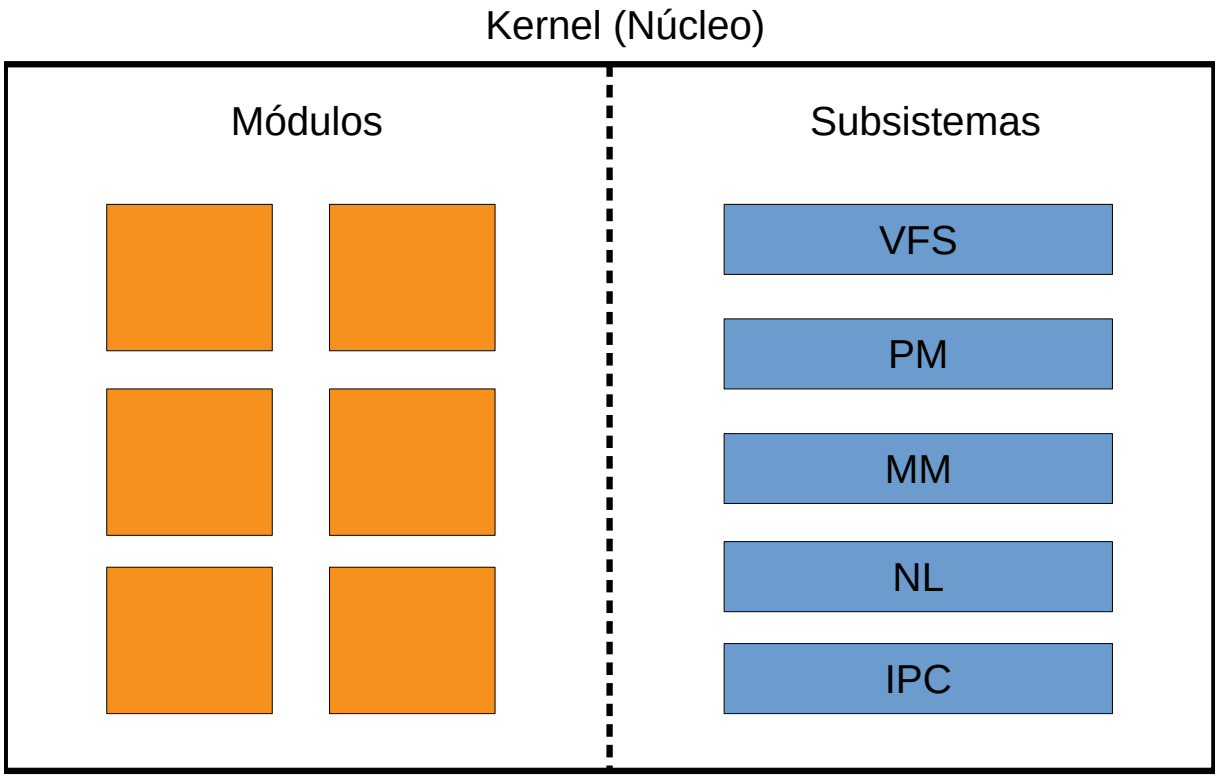


Figura 2.15: Composição de um *kernel*.

A Seção 2.2.3 apresentará o módulo de sistemas de arquivos virtual (VFS), mostrado na Figura 2.15, que é o sistema responsável pelo gerenciamento de sistemas de arquivo e ponto chave para a criação de sistemas de arquivo criptografados em espaço de usuário.

2.2.3 VFS

Sistemas de arquivos virtual, VFS da sigla em inglês, é um subsistema do *kernel* responsável por tratar com arquivos e sistemas de arquivos diversos. O VFS provê interfaces para os programas executados em modo usuário [23]. De maneira geral o VFS permite ao Linux dar suporte a múltiplos tipos diferentes de sistemas de arquivos [24], permitindo que vários sistemas de arquivo e tipos de mídia diferentes coexistam em uma mesma estrutura [5].

Um dos objetivos buscados com a criação do VFS é colocar uma extensa coleção de informações no *kernel* de modo a representar uma vasta quantidade de sistemas de arquivo, dessa maneira todos os sistemas de arquivos utilizados pelo Linux dependem do VFS, e são acessados pelo VFS através de sua interface inferior. O VFS possui interfaces padrão de leitura e escrita com os aplicativos de usuário [5], chamada de interface superior, criando uma camada de abstração que permite o acesso aos dados sem a necessidade de saber em qual tipo de sistema de arquivos o dado está armazenado, mesmo que sejam mídias diferentes, e simplificando a programação dessas aplicações [23][24]. A Figura 2.16, adaptada de uma figura maior encontrada em [7], mostra de maneira abstrata o funcionamento e conteúdo do VFS.

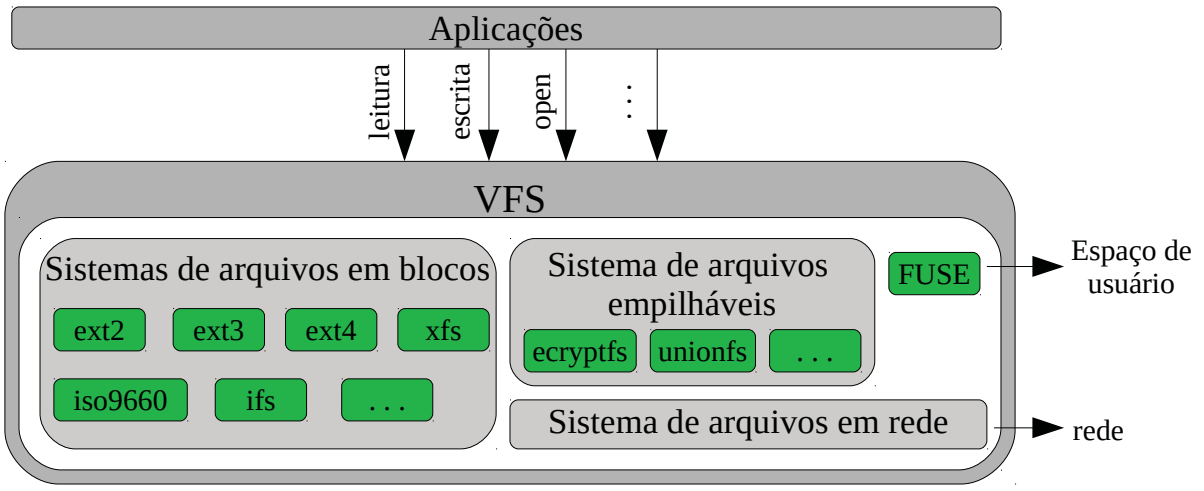


Figura 2.16: Componentes de um VFS, baseada em [7]

Na Figura 2.16 é possível ver que os sistemas de arquivos implementados e acessados pelo VFS variam bastante quanto a seu tipo e sua mídia utilizada para armazenamento, porém também se torna bastante claro que as aplicações dos usuários não precisam, nem possuem, acesso direto aos sistemas, fazendo apenas chamadas genéricas ao VFS. Dentre as categorias de sistemas de arquivo encontrados dentro do VFS se destacam os sistemas de arquivos em blocos que são os mais comuns e representam, de maneira geral, os sistemas locais armazenados e HDs. Também é importante notar os sistemas de arquivos empilháveis que são sistemas de arquivo que não tratam diretamente do armazenamento dos dados, e sim realizam algum pré processamento antes de enviarem os dados para um outro sistema de arquivo que realizará o armazenamento.

Outro ponto importante, não mostrado na figura, é a existência de um *cache* de páginas, *page cache*, na figura. De maneira geral nas operações de leitura os sistemas de arquivo, por meio da interface inferior do VFS, buscam a informação necessária, salvam no cache e retornam um ponteiro dizendo onde a informação foi salva. De maneira contrária porém análoga as operações de escrita devem primeiro ser salvas na *cache* e posteriormente ser enviado para o VFS a posição onde a informação deve ser salva, que por sua vez se comunicará com o sistema de arquivos que

buscará o dado na *cache* e colocará na posição correta. O *cache* de páginas possui como principal função agilizar os processos de leitura e escrita, realizando menos acessos aos dispositivos [14].

Uma outra parte importante mostrada na Figura 2.16 é o FUSE, sigla do inglês para *filesystem in userspace* se traduz para sistemas de arquivo no espaço do usuário. A interface FUSE permite a criação de sistemas de arquivo em espaço de usuário e é a base de implementação para o EncFS. A seção 2.2.4 abordará o tópico de forma mais aprofundada.

2.2.4 FUSE

Grande parte dos *kernels* monolíticos possuem implementados dentro deles, ou em módulos, os diversos sistemas de arquivos que podem ser utilizados, diminuindo o *overhead* da troca de mensagens com um programa no espaço de usuário, como ocorre com os *micro-kernels*. Os sistemas de arquivos em espaço de usuário tem, no entanto, crescido ao longo dos anos, impulsionados principalmente por trazerem algumas funcionalidades, sua facilidade de prototipação em estudos acadêmicos e o lançamento de mais e mais sistemas de arquivos desse tipo por um vasto número de empresas. Além disso muitos sistemas de arquivos já existentes em espaço de *kernel* foram adaptados para serem utilizados em espaço de usuário [25].

FUSE é atualmente a *framework* para criação de sistemas de arquivos em espaço de usuário mais utilizada no mundo [25]. De maneira geral o FUSE pode ser considerado como uma simples protocolo cliente servidor no qual o *kernel*, representado pelo módulo FUSE, é o cliente e o programa, criado pelo desenvolvedor do sistema de arquivo, em espaço de usuário é o servidor. Este programa em espaço de usuário é, geralmente, um *daemon*, sendo executado em *background* e aguardando as requisições do *kernel*. A Figura 2.17 mostra o funcionamento de um sistema utilizando FUSE.

No exemplo da Figura 2.17 um sistema de arquivos virtual criado utilizando FUSE é montado em um diretório. Quando se executa um comando na interface do sistema de arquivos a requisição é feita ao *kernel*, por meio do VFS, que, por sua vez, verifica que o sistema de arquivos utiliza FUSE e transfere a requisição para o módulo. O módulo FUSE então coloca a requisição em uma fila. O programa do usuário recupera a requisição da fila e realiza os processamentos necessários para recuperar a informação requisitada e responde, através do módulo FUSE, para o VFS que entrega a informação ao requisitante. Esse processo permite que, durante o processamento, tratamentos especiais possam ser dados aos arquivos de acordo com as necessidades de cada requisição [25], como por exemplo encriptação ou decriptação dos dados.

O módulo do FUSE foi adicionado ao *kernel* do Linux em sua versão 2.6.14, e está presente desde então, e para que seja realizada a comunicação do mesmo com o sistema de arquivos do usuário é utilizada a biblioteca *libfuse*.Embora possam ser utilizadas outras bibliotecas, ou até mesmo codificar as mensagens diretamente, a *libfuse* é a mais comum e largamente utilizada para este propósito, sendo a referência para outras implementações e sendo parte das maiores distribuições do Linux.

2.2.5 EncFS

EncFS é um programa que provê um sistema de arquivos criptografado executado em espaço de usuário utilizando FUSE. Criado em 2003 tinha como objetivo ocupar o espaço deixado por outros sistemas de arquivos criptografados baseados em NFS [26] ou implementados diretamente no *kernel*, que não conseguiram acompanhar o desenvolvimento do Linux [27].

O EncFS provê dois modos de encriptação distintos e que são utilizados em locais diferentes. O primeiro modo consiste em uma cifra de fluxo utilizando o CFB, conforme apresentado na Subseção 2.1.3.3, que é utilizado para encriptar o nome dos arquivos e os blocos

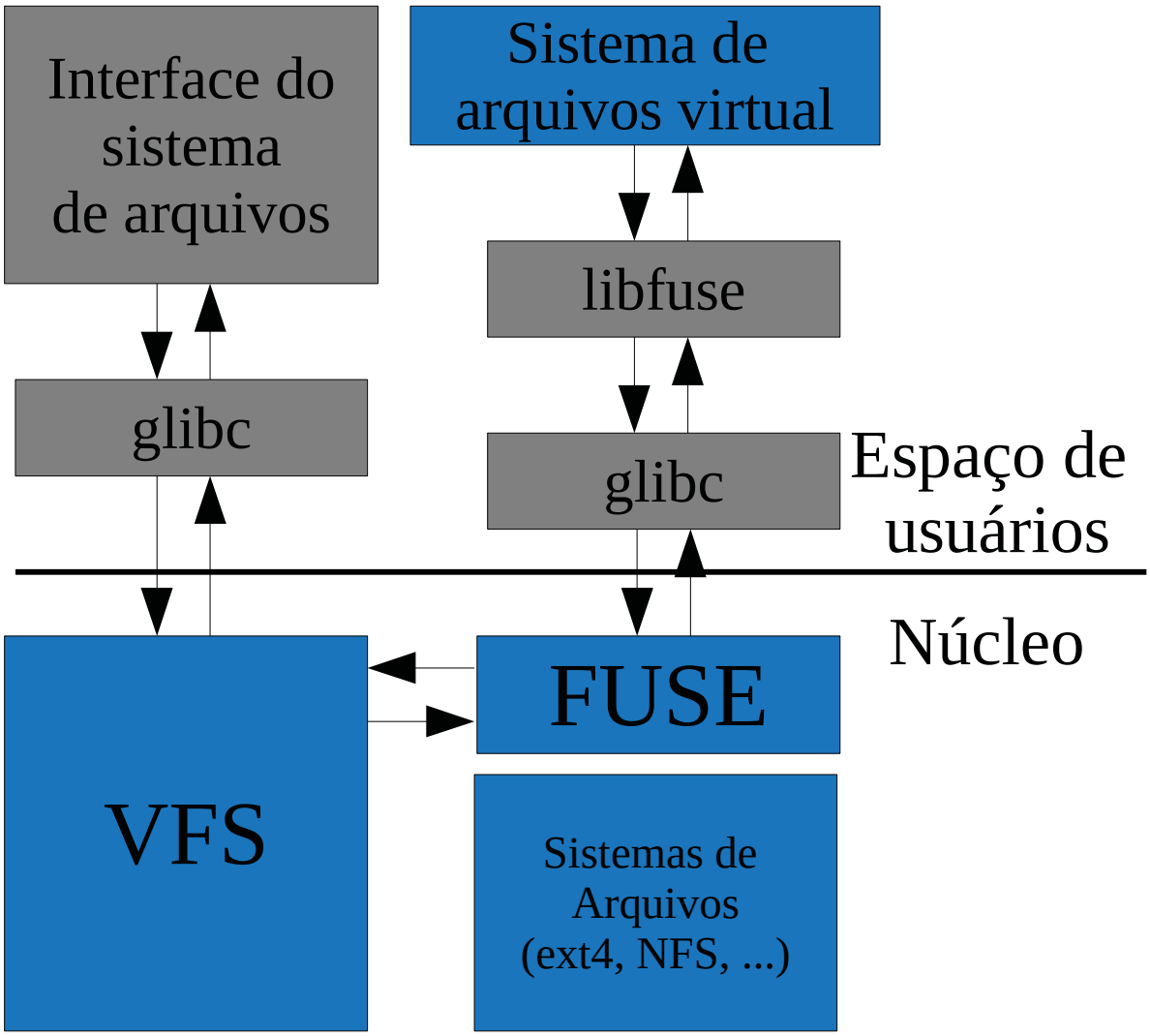


Figura 2.17: Exemplo de funcionamento de um sistema de arquivos utilizando FUSE [8].

incompletos no final dos arquivos. Este modo realiza várias passagens sobre os dados, invertendo a ordem em cada uma delas, para que possa aumentar a independência entre eles. O segundo modo é chamado de encriptação de bloco. Nesse modo são encriptados blocos do sistema de arquivos com tamanho fixo utilizando CBC, como apresentado na Subseção 2.1.3.2. O tamanho do bloco do sistema de arquivos utilizado pelo EncFS deve ser múltiplo do tamanho do bloco utilizado pelo cifrador e pode ter tamanho máximo de 4KiB. Cada bloco possui um IV determinístico para facilitar o acesso aleatório aos dados [27].

Durante a criação ou montagem do sistema de arquivos é requisitado do usuário a passagem de uma senha que é utilizada indiretamente para a encriptação e decriptação dos arquivos. A senha escolhida pelo usuário é utilizada para decriptar a chamada senha de volume, que por sua vez é usada nos processos envolvendo os arquivos, isso permite que um usuário troque sua senha de acesso sem a necessidade de processar novamente os arquivos criptografados. A senha de volume, por sua vez, é gerada a partir da utilização do algoritmo PBKDF2.

Todas as criptografias realizadas pelo EncFS utilizam os algoritmos disponibilizados pelo OpenSSL que são utilizados para encriptar tanto os arquivos quanto seus nomes. Durante a inicialização do sistema de arquivos é necessário informar dois diretórios, sendo um de origem e outro de destino. O diretório de origem é o local onde os arquivos criptografados serão

armazenados, retornando apenas resultados aleatórios caso um sistema ou usuário tente acessar os dados diretamente por ele. O diretório de destino, por outro lado, é onde será montado o sistema de arquivos do EncFS, quando um sistema ou usuário tenta acessar os dados no diretório de destino este informa o EncFS que por sua vez faz o processamento para encriptar ou decriptar a informação de acordo com o que for necessário [27].

O EncFS possui uma série de opções que podem fazer grande diferença durante a utilização do sistema de arquivos. A primeira e mais importante é a opção de escolha de cifra o que permite a utilização de diferentes cifras de bloco em seus processo, embora atualmente a grande maioria utilize AES. Outra opção é a escolha do tamanho do bloco que influencia diretamente no desempenho esperado do sistema. Uma opção muito importante, que vem ativada por padrão nas versões atuais do EncFS, é a utilização de um IV diferente por arquivo. Nas primeiras versões cada bloco de uma arquivo possuía um IV diferente, porém eram escolhidos de forma determinística de modo que era possível identificar se dois arquivos eram idênticos analisando suas cifras. Além dessas opções apresentadas existem muitas outras que trazem versatilidade e possibilidades de personalização dos sistemas de arquivos. Estas opções também trazem uma maior segurança aos dados armazenados uma vez que a ativação de algumas delas previne a utilização de algumas formas de ataque [27].

2.2.6 Sistemas de arquivos em espaço de *kernel*

Além dos sistemas de arquivos criptografados que são executados em espaço de usuário, como é o caso do EncFS, também existem aqueles que são executados em espaço de *kernel*. Sistemas de arquivos criptografados em espaço de *kernel* possuem como vantagens a necessidade de um número menor de trocas de contexto para processar requisições, o que pode levar a um aumento de desempenho, e maior transparência em relação aos processos que estão sendo executados, podendo verificar e executar operações que são impossíveis em espaço de usuário, por outro lado o processamento em espaço de *kernel* exige um nível de permissão maior para que possa ser executado e, de maneira geral, possui uma complexidade maior para ser implementado. Alguns exemplos de sistemas de arquivos criptografados em espaço de *kernel* são eCryptfs, dm-crypt, ZFS e fscrypt.

O eCryptfs [28] é um exemplo de sistema de arquivos criptografado em espaço de *kernel* no Linux. Por ser um sistema de arquivos empilhado necessita de um sistema de arquivos suporte que consiga se comunicar com o dispositivo de armazenamento. Uma de suas características é o fato de manter os dados relativos à encriptação de um arquivo dentro dos metadados do próprio arquivo, permitindo desta forma que este arquivo possa ser copiado para outra máquina onde pode ser decriptado utilizando a chave correta. Este sistema é utilizado como base para o sistema de encriptação do diretório *home* do Ubuntu além de ser nativo no ChromeOS e em diversos sistemas de armazenamento em rede.

Outro exemplo de sistema de encriptação com processamento em espaço de *kernel* é o dm-crypt [29]. No Linux um *device-mapper* é um sistema que permite a criação de camadas virtuais de dispositivos de blocos, dessa forma o dm-crypt, sigla para *device-mapper crypt*, permite a encriptação diretamente desses dispositivos. Como a encriptação é realizada diretamente nos dispositivos de armazenamento o dm-crypt não é em diretamente um sistema de arquivos, porém é possível montar um sistema de arquivos no dispositivo encriptado de maneira transparente, dessa forma encriptando todos os dados e metadados do mesmo. De maneira mais geral o que pode ser feito em um dispositivo não criptografado pode ser feito de maneira transparente em um dispositivo encriptado com o dm-crypt.

Criado inicialmente pela Oracle, para funcionamento em seu sistema operacional Open Solaris, o sistema de arquivos ZFS [30] foi posteriormente modificado para ser um módulo do

kernel do Linux. Com o tempo, devido ao código deixar de ser aberto um ramo alternativo foi criado para manter o desenvolvimento para outros sistemas operacionais o que gerou a criação do projeto *Open ZFS* [31], que é responsável pela versão atual, que não é mantido juntamente com o *kernel* do Linux devido a incompatibilidade de licenças. Como o desenvolvimento do sistema de arquivos ZFS não possui como foco a encriptação dos dados, e sim outros atributos como facilidade de gerenciamento e escalabilidade, ele sofre com uma grande perda de desempenho.

Na versão 4.1 do *kernel* do Linux foi introduzida a criptografia nativa do sistema de arquivos Ext4. Para o acesso às funcionalidades de criptografia do módulo Ext4 é utilizada a biblioteca fscrypt [32], como é comumente conhecido o processo de encriptação como um todo, que é uma biblioteca capaz de realizar a encriptação de arquivos e diretórios de forma transparente. O fscrypt se destaca por ser uma criptografia a nível de arquivo, não encriptando completamente os blocos do disco, e simultaneamente não ser um sistema de arquivos empilhado.

2.3 EXTENSÕES AVX E INSTRUÇÕES AES-NI

A busca por maior performance e eficiência na hora de processar os dados levou as fabricantes de processadores a chegar aos limites impostos pela física em termos de frequência de *clock* utilizadas. Diante desta barreira foi necessário buscar melhorias de outras formas. Neste contexto surge o paralelismo nas CPUs que permitiu que um processador com o mesmo *clock* realizar um número muito maior de tarefas.

O paralelismo pode ser implementado de diversas formas diferentes. Uma classe muito importante quando se fala de CPUs é a chamada *single instruction multiple data* (SIMD), que é o método no qual é utilizada uma única instrução porém aplicada simultaneamente a diversos dados diferentes, gerando múltiplos resultados de uma única vez. A Figura 2.18 ilustra o funcionamento de uma instrução SIMD quando comparada com uma instrução escalar.

Embora já fossem implementadas em CPUs anteriores as extensões SIMD sofreram uma grande melhoria com a introdução das extensões AVX. Além disso o aumento na utilização da cifra de blocos AES levou a criação das AES-NI para processamento eficiente deste algoritmo de cifragem. As seções 2.3.1 e 2.3.2 falam, respectivamente, sobre o funcionamento das extensões AVX e das instruções AES-NI.

2.3.1 AVX

Introduzidas pela Intel em 2008 a extensão AVX é um conjunto de instruções para a realização de processamento SIMD [9]. Sendo projetadas para substituir as instruções SIMD anteriores, chamadas *Streaming SIMD Extensions* (SSE) [33], possuem algumas características que as tornam melhores que as versões anteriores. A primeira e mais simples característica, apesar de ser uma das mais importantes, é o tamanho dos registradores. Nas versões antigas do SIMD os registradores possuíam 128 bits, enquanto no AVX foram expandidos para 256 bits com suporte para ampliação no futuro [9], sendo lançado em 2017 a nova versão do AVX com registradores de 512 bits [34]. Outra característica importante é sua capacidade de realizar operação com três operandos de forma não destrutiva, conseguindo o resultado enquanto mantém os valores de entrada intactos. As demais características estão relacionadas com maior facilidade de utilização e possibilidades futuras.

As instruções AVX funcionam com uma gama de tipos diferentes de dados, embora nem todas as instruções funcionem com todos os tipos. De maneira geral são suportados os tipos de ponto flutuante tanto de precisão simples quanto de precisão dupla e, assim como no antigo SSE, possui suporte a inteiros de 8, 16, 32, 64 e 128 bits [9].

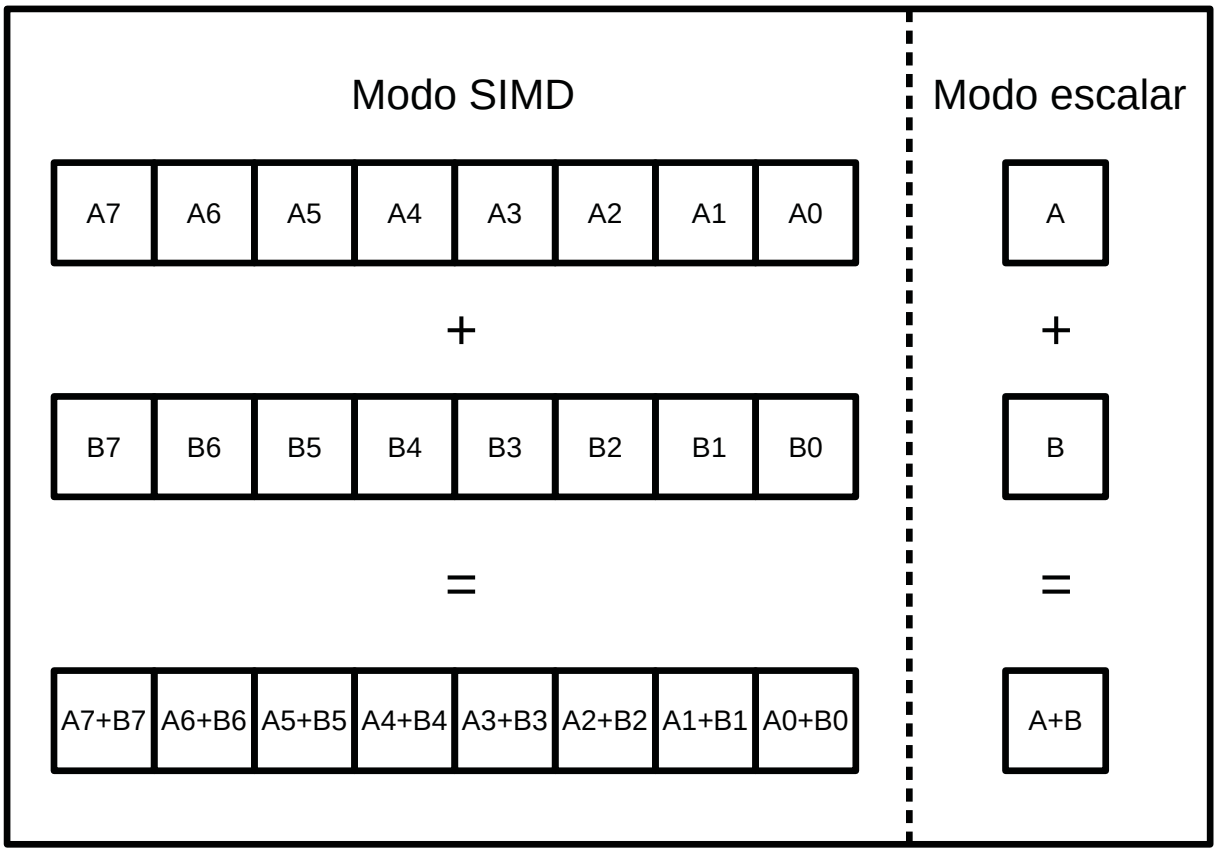


Figura 2.18: Exemplo de instrução SIMD e escalar, baseado em [9].

Para dar suporte ao AVX a implementação de hardware consiste em 16 registradores de 256 bits chamados de YMM além de um registrador de controle de 32 bits. De modo a manter a compatibilidade com as instruções SSE antigas os registradores são alinhados de modo que os 128 bits menos significativos de cada um dos registradores YMM pode ser visto como um registrador do tipo XMM, utilizado pelo SSE [9].

A memória é dita alinhada quando os dados que se deseja carregar para os registradores estão armazenados de forma contígua. De maneira geral as instruções SSE necessitavam que a memória estivesse alinhada para que pudesse ser carregada para o registrador. O AVX relaxa essa necessidade porém memórias não alinhadas podem causar perda de performance, desta forma ainda é uma boa prática manter a memória alinhada. Outra parte importante no que diz respeito à performance é a utilização de instruções antigas junto com instruções novas que causa perda de desempenho e devem ser evitadas [9].

De maneira geral as instruções AVX, mesmo em sua primeira versão, geram ganhos de desempenho de até duas vezes [9]. Alguns anos depois do lançamento do AVX foi lançado o AVX2, que trouxe instruções que realizavam multiplicação e soma de forma direta melhorando precisão e desempenho, e o AVX512, que dobrou o tamanho dos registradores. Em ambientes Linux é possível ver se existe suporte as instruções AVX utilizando o comando *cat /proc/cpuinfo* e verificando se a CPU apresenta a *flag* avx ou avx2.

2.3.2 AES-NI

Seguindo a linha que busca aprimorar a performance de suas CPUs em tarefas que são realizadas muitas vezes a Intel criou as AES-NI, sigla para *AES new instructions*, em 2010 que

tem como objetivo permitir a encriptação e decriptação de dados utilizando o AES de forma rápida e segura [35].

O conjunto AES-NI é composto por seis instruções totalmente implementadas em hardware. Essas instruções são então divididas em dois grupos, um com quatro instruções responsável por encriptar e decriptar os dados e outro com duas instruções responsável pela expansão da chave. Estas instruções podem ser utilizadas em todas as variantes do AES, incluindo todos os tamanhos de chave, o modo padrão de funcionamento e ainda algumas variantes que não seguem o padrão [35].

As quatro instruções utilizadas para encriptação e decriptação são chamadas *AESENC AESENCLAST AESDEC* e *AESDECLAST* e se referem a encriptação de uma rodada, encriptação da última rodada, decriptação de uma rodada e decriptação da última rodada, respectivamente. Cada uma dessas instruções realiza um conjunto de sequências de transformação do AES, seja de encriptação ou decriptação [35]. Essas funções permitem que os algoritmos de encriptação e decriptação sejam implementados de forma simples, evitando possíveis erros no código utilizado.

As outras duas funções apresentadas são chamadas *AESKEYGENASSIST* e *AESIMC* e são utilizadas na etapa de expansão da chave. Enquanto *AESKEYGENASSIST* é utilizada diretamente para a expansão da chave principal nas diversas chaves utilizadas nas rodadas a função *AESIMC* faz a conversão das chaves da rodada para uma forma utilizável na decriptação [35].

Em termos de performance, quando são analisadas as implementações dos modos de encriptação do AES que podem ser realizados em paralelo, como no caso do CTR, é possível ver um ganho de performance superior a dez vezes, em alguns dos casos, quando comparado com as implementações baseadas em tabelas feitas em *software*, que são as implementações mais rápidas feitas em *software*. Em implementações de modos em que não é possível paralelizar as partes, como é o caso do modo CBC, ainda é esperado um ganho de desempenho entre duas e três vezes maior que as melhores implementações em *software*[35].

2.4 MÉTODOS DE AVALIAÇÃO PARA SISTEMAS DE ARQUIVOS

Sistemas de arquivos podem ser avaliados de diversas maneiras diferentes de modo a destacar características diferentes. É bastante comum avaliações nas quais é verificado os gastos de CPU e memória durante a utilização do sistema no entanto os testes mais utilizados são os que verificam a latência, número de operações de I/O e a vazão de leitura e escrita. Para a realização dos testes existem programas especializados que podem realizar testes de estresse, quando se verifica o limite máximo atingido, ou testes de simulação, quando é simulado o comportamento de algum tipo de aplicação. A seguir são apresentados os programas Fio, Iozone e Filebench que são utilizados para avaliação de performance de sistemas de arquivos.

Fio, sigla do inglês para *flexible I/O tester* ou testador flexível de I/O, foi criado com o objetivo de se evitar a necessidade de criação de um programa de teste específico sempre que se buscava verificar uma carga diferente [36], por esse motivo Fio foi criado de forma bastante versátil e adaptável para diversos testes. Para o funcionamento do programa é necessário criar um arquivo de trabalho no qual é descrito como será realizado o teste, definindo uma gama enorme de parâmetros o que permite testes precisos para os casos desejados.

IOzone, por outro lado, é focado em testes de estresse, sendo uma ferramenta de testes que gera medidas para uma gama de operações [37]. Focado em criar um perfil geral do sistema de arquivos o programa realiza diversas operações de leitura e escrita com parâmetros diferentes e, ao final, apresenta gráficos dos resultados obtidos. Por não simular a utilização de uma aplicação não é possível estimar a performance da aplicação desejada, porém com os resultados gerais é

possível averiguar quais tipos de aplicações seriam mais adequadas para serem utilizadas no sistema testado.

Outro exemplo de programa que realiza simulação de funcionamento de aplicações é o Filebench. Assim como o Fio o Filebench é bastante versátil, permitindo serem especificados o comportamento de quais aplicações se deseja testar [16], porém sendo menos engessado pois suas cargas de trabalho não são inseridas diretamente no código [38], permitindo não somente uma alteração de parâmetros dos testes como também o comportamento específico do teste em si, podendo gerar padrões complexos. O comportamento das aplicações é definido em arquivos utilizando a linguagem WML, sigla do inglês para *workload model language*, que especifica diferentes processos, com diferentes números de *threads*, e indica quais operações serão realizadas pelo processo. No artigo *Filebench: A flexible framework for file system benchmarking*[38] é apresentado o funcionamento da linguagem WML e mostra como é possível realizar customizações para simular sistemas bastante distintos com peculiaridades e necessidades muito diferentes. Durante a instalação o Filebench realiza o download de diversas cargas de trabalho predefinidas que não são recomendadas para utilização direta, pois tentam dar uma ideia geral podem não representar a utilização do sistema que se deseja testar, mas que ajudam na criação de cargas de trabalho mais específicas. Ao final dos testes são apresentados os resultados para diversas medidas diferentes como vazão, latência e número de operações por segundo [38].

2.5 ASPECTOS PRINCIPAIS

Nas seções anteriores foram apresentados de forma abrangente os tópicos da fundamentação teórica, porém para cada um possui aspectos principais para seu entendimento.

Na Seção 2.1 é importante entender que o AES é uma cifra de blocos e, como consequência, só consegue encriptar blocos de 128 bits. Para que consiga encriptar dados com mais de 128 bits é necessário utilizar modos de encriptação. Destes o principal, que foi utilizado no trabalho, é o modo CTR, uma vez que pode ser paralelizado e pré-processado.

A Seção 2.2 apresenta a estrutura do Linux e possui como aspectos mais importante a diferenciação entre os modos núcleo e usuários, bem como a função do subsistema VFS e o funcionamento do módulo FUSE. Estes tópicos se relacionam diretamente com o EncFS++ uma vez que sua encriptação é realizada em modo usuário utilizado o módulo FUSE.

As extensões AVX e as instruções AES-NI são apresentadas na Seção 2.3. É importante entender que as extensões AVX permitem a aceleração de processos paralelizáveis utilizando processamento SIMD e que as instruções AES-NI utilizam hardware específico para conseguir grandes ganhos de desempenho no processamento do AES. Uma vez que a EncFS++ é paralelizável e utiliza o AES essas tecnologias permitem grandes ganhos de desempenho.

A Seção 2.4 apresenta diversos métodos para avaliação de sistemas de arquivos, sendo o principal utilizado neste trabalho a ferramenta Filebench que realiza testes de simulação de carga.

3 TRABALHOS RELACIONADOS

A busca por melhorias de performance dos processos de encriptação e decriptação de dados assim como nos processos relacionados a sistemas de arquivos já levou a realização de muitas pesquisas nas respectivas áreas. Muitas destas pesquisas levaram a implementação otimizadas de algoritmos e desenvolvimento de técnicas de modo a explorar ao máximo os recursos computacionais disponíveis.

No artigo *Implementation and analysis of aes encryption on GPU*[39] foi realizada a implementação do algoritmo AES e o estudo de algumas variáveis que poderiam afetar sua performance quando utilizado de maneira paralela. Embora seu foco tenha sido completamente nas implementações em GPU parte dos resultados encontrados podem ser extrapolados para se ter um comportamento esperado em CPU.

Estudos sobre implementações do AES para os extremos de CPUs de baixa capacidade, microcontroladores de 8 bits, e GPU foi realizado e apresentado no artigo *Fast software aes encryption* [40], enquanto estudos para sistemas intermediários, processadores i7, e GPUs foram apresentados em *High-performance symmetric block ciphers on multicore CPU and GPUs* [41]. Ambos os estudos mostraram ser possível extrair ao máximo a vazão em CPU. Em alguns casos foram obtidos resultados em que a vazão do processamento em CPU foi maior que o do processamento em GPU, devido a utilização da tecnologia AES-NI e das necessidades de envio de dados do sistema principal para a GPU, que geram perdas de desempenho.

Outros trabalhos também exploraram diferentes implementações do algoritmo AES tanto para CPU como para GPU. Um exemplo é o artigo *Different implementations of aes cryptographic algorithm*[42] no qual foi comparada a implementação FAST do AES com as implementações utilizando AES-NI e GPU com CUDA. Os experimentos mostraram que a implementação FAST foi aproximadamente 20 vezes mais lenta que a implementação em CUDA para GPU que, por sua vez, foi 2 vezes pior que a implementação utilizando AES-NI.

Uma nova técnica para a realização de encriptação foi apresentada em *Parallel speculative encryption of multiple aes contexts on GPUs*[15]. Após a implementação de uma nova versão de AES paralelizada em GPU, que possui uma pequena modificação de modo a evitar problemas de extremidade entre os dados entre o sistema principal e a placa de vídeo, é mostrado o modo de encriptação especulativa. A encriptação especulativa se utiliza da capacidade de pré-processamento de contextos do modo de operação CTR, gerando máscaras de forma paralela que já estão prontas quando o dado precisa ser encriptado. O trabalho mostra que a nova técnica possui vazão bastante superior a outros trabalhos.

Também ocorreram várias pesquisas relacionadas a sistemas de arquivos. Inicialmente apresentado em *Performance and extension of user space file systems* [43] foi realizada a comparação entre diferentes implementações de sistemas de arquivos em espaço de usuário, criados em linguagens diferentes, com o sistema de arquivos ext3 nativo. Como mostrado as implementações variam um pouco e a performance, de maneira geral, diminui um pouco mas a viabilidade do sistema depende bastante da carga que será utilizada.

São realizados estudos similares em *Terra incognita: On the practicality of user-space file systems* [44] e *Performance and resource utilization of fuse user-space file systems* [45] porém com foco menor em diferentes implementações e maior em diferentes tipos de carga de trabalho utilizadas pelos sistemas. Ambos os estudos, assim como o apresentado anteriormente chegam a conclusão de que existe uma perda inerente no uso de sistemas de arquivos em espaço de usuário,

porém esta perda é pequena no caso geral. Os estudos também verificaram que, apesar do caso geral, a perda de desempenho sofre grande variação de acordo com o tipo de carga aplicado a ele.

Com o inicio dos estudos sobre sistemas de arquivos criptografados buscou-se implementar padrões que facilitassem o uso dos recursos disponíveis. O chamado *OpenBSD cryptographic framework* (OCF) [46] foi um dos resultados desta busca. Sendo criado para ser uma interface comum para diferentes placas aceleradoras que permitisse o uso deste recurso de forma uniforme e eficiente. Foi mostrado que o sistema consegue atingir até 95% de performance do limite teórico do equipamento. O OCF foi adaptado para uso em diversos outros sistemas operacionais, como FreeBSD e Linux.

Em *A file system using GPU-accelerated file-wise reliability scheme* [47] é revisado uma técnica para a criação de um sistema de arquivos acelerado por GPU e esquema de confiabilidade por arquivo apresentado em um trabalho anterior do mesmo grupo. Posteriormente em *A reliable and secure GPU-assisted file system* [48] foi apresentado um trabalho similar no qual o sistema é expandido de modo a possuir não apenas a confiabilidade mas também trazendo segurança para os arquivos dentro dele.

Baseado na técnica de encriptação especulativa é apresentado em *Speculative encryption on GPU applied to cryptographic file systems* [49] um sistema de arquivos criptografado com aceleração em GPU. Sendo implementado como uma adaptação do sistema de arquivos criptografado EncFS o novo sistema, chamado EncFS++, utiliza janelas de contexto. Quando a requisição é feita para encriptação ou decriptação de um arquivo é preparada a janela que contém contextos com valores de contador em sequência. O sistema de arquivos então requisita à biblioteca a preparação de todos os contextos existentes na janela utilizando seus respectivos contadores. Após utilizar um contexto para a encriptação ou decriptação de dados a janela desliza de modo que o contexto utilizado seja descartado e um novo contexto possa ser iniciado com um novo contador na sequência. Caso o acesso ao arquivo não seja feita de forma sequencial os contextos da janela são atualizados de forma que representem a nova posição do arquivo. Após o processo de preparação dos contextos quando uma parte do arquivos necessita ser encriptado ou decriptado o sistema apenas realiza um processo de XOR entre a parte e o contexto, caso este já tenha sido processado. Para o processamento dos contextos e encriptação dos blocos é utilizada a biblioteca WAESlib. Os testes realizados comparando a original EncFS com a nova EncFS++ mostram um aumento de performance de mais de 300% em alguns casos além de também apresentar ganhos quando comparado a outros sistemas de encriptação.

Este trabalho mostra a viabilidade da encriptação especulativa em CPU, que já foi provada para os casos em GPU. Os resultados mostram que a técnica de encriptação especulativa gera ganhos de desempenho no acesso aos dados armazenados, mesmo quando aplicados em testes que simulam ambientes com baixo poder de processamento. A encriptação especulativa utilizada em CPU apresentou ganhos de desempenho mesmo utilizando as implementações mais rápidas do AES.

4 IMPLEMENTAÇÃO E TESTES DA BIBLIOTECA WAESLIB CPU

Antes quer fosse realizada a implementação do sistema de arquivos era necessário validar o funcionamento da encriptação especulativa com processamento em CPU. Já tendo sido validada a versão em GPU os primeiros passos necessários eram os de implementação da biblioteca WAESlib em sua versão CPU e verificar a corretude e eficiência do funcionamento da mesma. Será apresentado na Seção 4.1 o funcionamento geral da biblioteca WAESlib e na Seção 4.2 será mostrado como a biblioteca foi adaptada para funcionamento em CPU.

4.1 WAESLIB

A biblioteca WAESlib foi inicialmente apresentada em *Parallel speculative encryption of multiple aes contexts on GPUs* [15] e era inicialmente implementada para realizar criptografia especulativa em GPUs utilizando a plataforma CUDA. Embora sua implementação inicial tenha sido voltada para utilização com uma tecnologia específica de processamento em GPUs a interface apresentada pelo biblioteca permite que esta possa ser adaptada para realizar todo processamento necessário em CPUs, onde pode utilizar algumas tecnologias para melhorar sua performance sem que esteja profundamente vinculada a nenhuma delas.

A interface apresentada pela WAESlib é composta de seis funções, sendo elas WAES_init, WAES_setkey, WAES_ctx, WAES_encrypt, WAES_decrypt e WAES_finish. Cada uma das funções deve desempenhar tarefas específicas que, de maneira geral, são comuns a qualquer implementação que seja feita da biblioteca. A seguir serão apresentadas as funções na sequência em que devem ser utilizadas para que a biblioteca funcione de maneira correta.

A primeira função que deve ser utilizada, quando se desenvolve um sistema que fará uso da WAESlib, é a função WAES_init. Esta função é a responsável por realizar todas as inicializações necessárias para o funcionamento correto das demais funções. A WAES_init deve alocar os espaços necessários para o armazenamento dos contextos, inicializar algumas variáveis necessárias e realizar as configurações e inicializações necessárias para uso das *threads* trabalhadoras. Após o fim das inicializações a função deve retornar um valor indicando se todos os procedimentos foram realizados sem erros para que as outras funções possam ser utilizadas.

Após a inicialização do ambiente é necessário utilizar as função WAES_setkey. De maneira geral uma chave é utilizada muitas vezes e seria desnecessariamente custoso realizar a expansão da chave todas as vezes que se deseja realizar uma encriptação. Desta forma a função WAES_setkey é responsável por armazenar a chave informada bem como sua forma expandida de modo que, quando for necessário utilizá-la basta informar seu índice. Ao ser chamada a função também recebe qual o tamanho da chave esperado para que possa verificar se a chave possui o tamanho correto e expandir para o número certo de rodadas.

Com o ambiente inicializado e as chaves expandidas e armazenadas é possível começar a etapa de inicialização dos contextos. A criptografia especulativa possui a característica de criar os contextos necessários de forma antecipada para que possam ser utilizados quando necessário e a função WAES_ctx é a responsável por agendar o processamento desses contextos. A função recebe um número de contexto, o índice de uma chave já cadastrada e um IV e agenda o processamento desse contexto utilizando a chave e o IV informados. O processamento desses contextos é realizado em paralelo, podendo ser processados múltiplos contextos simultaneamente.

Com os contextos prontos o ambiente está preparado para a realização da encriptação ou decriptação dos dados. As funções WAES_encrypt e WAES_decrypt possuem essa função.

Como a WAESlib utiliza o modo CTR as funções de encriptação e decriptação são iguais e apenas realizam um XOR entre o contexto e o dado que se deseja encriptar, como mostrado na Seção 2.1.3.5, por esse motivo as funções WAES_encrypt e WAES_decrypt são idênticas. De maneira geral o tempo para transferência de dados para GPU não compensa a realização do XOR mais rápido realizado no dispositivo, dessa forma mesmo em implementações em GPU essas funções são processadas em CPU.

Ao término dos processamento dos dados é necessário liberar os recursos reservados para a aplicação. Desse modo a função WAES_finish possui, de certa forma, atribuições contrárias às da função WAES_init, sendo responsável pela liberação dos recursos alocados.

4.2 IMPLEMENTAÇÃO DA CRIPTOGRAFIA ESPECULATIVA EM CPU

Durante a primeira fase do trabalho foi realizada a implementação da biblioteca WAESlib para realização do processamento em CPU. Seguindo o funcionamento apresentado na Seção 4.1 cada função foi codificada e testada de modo a evitar erros diminuir ao máximos seus tempos de processamento. Além das funções é necessário armazenar duas listas em memória RAM, uma para o armazenamento das chaves que serão utilizadas para a realização da encriptação e outra onde serão armazenados os contextos criptografados.

Para o armazenamento das chaves foi implementada um vetor que cresce conforme a demanda. Este vetor dinâmico possui um desempenho um pouco menor que um lista estática quando se armazena uma chave nova, porém como a inserção de chaves ocorre poucas vezes esta diferença pode ser ignorada. Para o caso dos contextos o programador que utiliza a biblioteca informa o número máximo de contextos, de 4KiB ou 8 KiB cada, que deseja e é criado um vetor estático na memória para que possam ser armazenados.

A função WAES_init é implementada para receber como entrada o número máximo de contextos e o tamanho de cada contexto. Com os valores recebidos a função cria o vetor de chaves e o vetor de contextos, inicializando os valores das variáveis de controle de cada um dos contextos. Por fim cria filas de prioridade que serão utilizadas para alimentar as *threads* e inicializa as respectivas *threads*. Na versão atual o número de *threads* trabalhadoras é pré-definido dentro da biblioteca e não pode ser alterada pelo programador durante seu uso.

As *threads* são criadas utilizando *pthreads* e cada uma delas executa uma função que verifica quando existem elementos em sua fila e realiza a encriptação do contexto informado, utilizando a chave informada, pelo primeiro elemento da fila. As encriptações são realizadas utilizando a *libcrypto*, que é parte do conjunto *OpenSSL*. Para evitar que *threads* diferentes realizem alterações simultâneas num mesmo contexto são utilizados vários *locks* que causam grande impacto no desempenho da função mas que são necessários para o seu funcionamento correto. Diferente do processamento em GPU, as *threads* não podem ficar sempre verificando se existem elementos na fila pois acarretaria em uma alta utilização da CPU e lentidão do sistema sem nenhum resultado efetivo, por esse motivo as *threads* são colocadas em modo de espera quando verificam que sua fila está vazia e se mantém nesse estado até que sejam acordadas pela função WAES_ctx.

Existem três tarefas que são executadas pela função WAES_setkey. Primeiramente a função verifica se o vetor onde são armazenadas as chaves precisa ser aumentado e, caso precise, realiza a ampliação do vetor. Em seguida a função verifica se o número informado para a chave já existe, para que possa alterar uma chave já existente em vez de criar uma nova. A terceira tarefa é encontrar o ponto, já existente ou novo, onde a chave será armazenada e armazená-la, bem como informações sobre seu tamanho para que possam ser informadas à *libcrypto*.

A WAES_ctx é um das funções mais críticas da biblioteca, sendo responsável por agendar o processamento dos contextos. Possui como entrada o número do contexto que será processado bem como o número da chave que será utilizada, além disso recebe o IV e a prioridade do contexto. A função verifica se o contexto e a chave informadas são válidas e verifica se o contexto já está em alguma fila para processamento ou se precisa ser adicionado em uma. Caso o contexto já esteja em uma fila ele é apenas alterados e a fila de prioridade em que está é atualizada. Nos demais casos a função verifica qual a próxima fila candidata a receber o contexto e se ela não está cheia. Nos casos de fila cheia é verificada a próxima fila candidata até encontrar alguma com espaço. O tamanho máximo das filas que alimentam as *threads* é definido na biblioteca e não pode ser alterada pelo programador. Por fim, as variáveis de controle do contexto são alteradas, o contexto é colocado na fila escolhida e a *thread* correspondente é acordada. A função WAES_ctx também sofre com o grande uso de *locks*, que diminuem sua eficiência mas garantem seu funcionamento correto.

Como no modo CTR a encriptação e decriptação são iguais as funções WAES_encrypt e WAES_decrypt são implementadas para que internamente chamem uma mesma função, sendo esta função crítica para o bom desempenho da biblioteca. Possuem como entrada o número do contexto que será utilizado, o endereço do *buffer* quer será utilizado no XOR e quantos bytes serão encriptados ou decriptados. Inicialmente a função verifica se o contexto é válido e se o número de bytes indicados é menor que o tamanho do contexto. Em seguida a função verifica se o contexto já está pronto pra ser utilizado e aguarda caso não esteja, até um limite máximo definido pela biblioteca. Quando o contexto estiver pronto é realizado o XOR entre ele e o buffer. Está função também se utiliza de *locks* para garantir que o contexto não seja alterado enquanto está sendo utilizado.

A função WAES_finish por fim é responsável por liberar os recursos alocados inicialmente. Primeiro a função faz a liberação do espaço reservado para o armazenamento das chaves, depois finaliza as *threads* trabalhadoras e libera o espaço reservado para os contextos. A Figura 4.1 apresenta as ligações entre as diversas partes da WAESlib CPU.

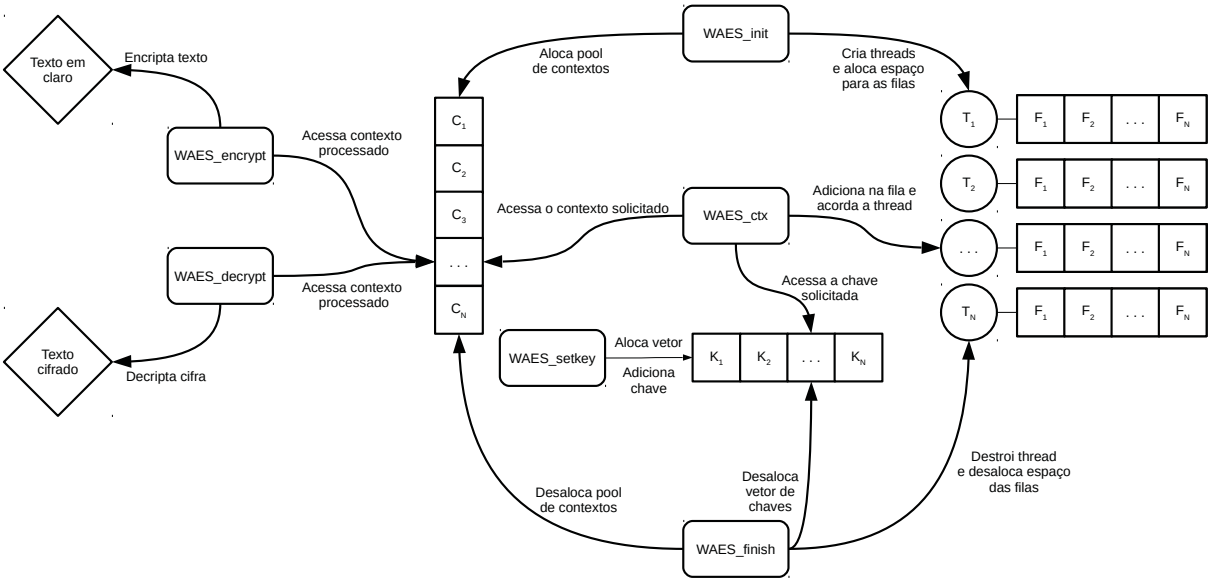


Figura 4.1: Ligações entre as diferentes partes da WAESlib CPU

Durante a implementação da biblioteca foram realizados diversos teste de caixa branca para verificar o desempenho das funções. As funções WAES_init, WAES_setkey e WAES_finish não possuem muito impacto durante a encriptação e decriptação dos dados por serem utilizadas

um número muito mais limitado de vezes quando comparado com as demais funções. Desta forma os testes focaram nas funções WAES_ctx, WAES_encrypt e WAES_decrypt. Os testes foram realizados em um computador com processador Intel Core i7-10700 com *clock* de 2,9GHz e 8 núcleos, que suporta 16 *threads* simultâneas, e com 16GB de RAM.

A Figura 4.2 apresenta testes realizados, onde foram feitas simulações de requisições que realizam acesso sequencial a um arquivo, onde o próximo bloco a ser encriptado é o próximo bloco do arquivo. O teste simulou o processamento realizado por 5 *threads* trabalhadoras, variando o número de *threads* clientes. A Figura 4.3 mostra o resultados para testes simulados com padrão de acesso aleatório, não existindo correlação entre o bloco atual e o próximo a ser processado. O teste foi realizado utilizando 1 *thread* cliente e variando o número de *threads* trabalhadoras. Nos gráficos também são apresentados os resultados dos testes utilizando a Libcrypto diretamente, removendo a chamada da WAES_ctx e substituindo a WAES_encrypt pela chamada da Libcrypto.

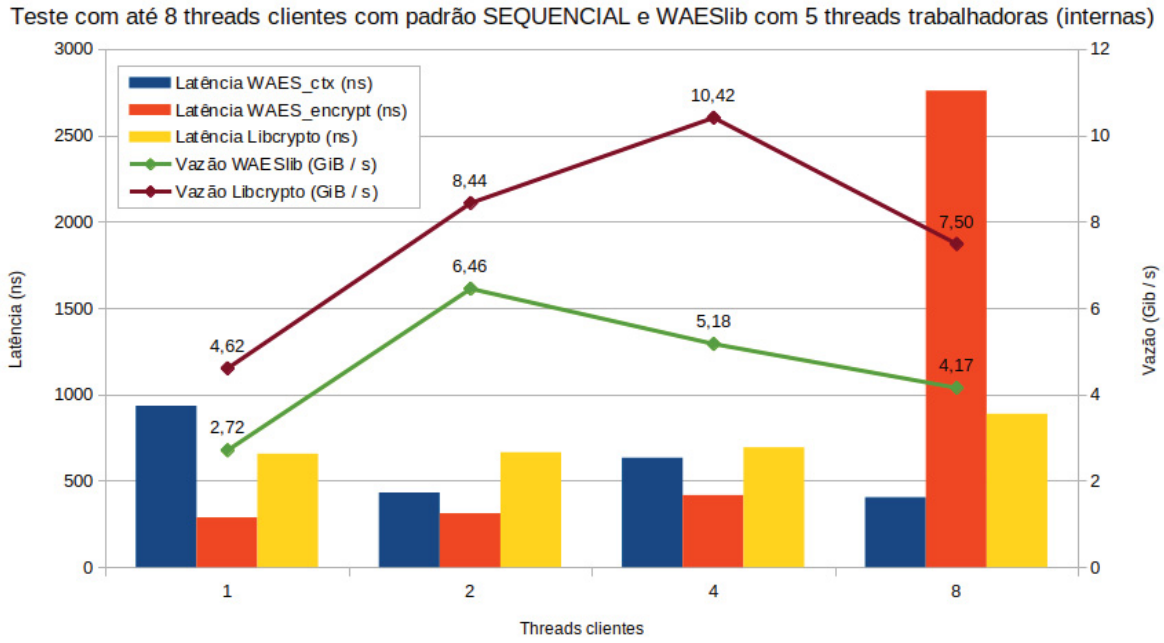


Figura 4.2: Resultado do teste de requisição sequencial variando o número de *threads* clientes.

Como é possível ver na Figura 4.2 as vazões alcançadas com a utilização da WAESlib foram menores do que as alcançadas utilizando diretamente a Libcrypto. Parte deste resultado se da pelo fato da Libcrypto não se utilizar de *threads* trabalhadoras, evitando trocas de contexto, porém parte pode ser devido à implementação da WAESlib. Durante a segunda fase do trabalho foi realizada a otimização do sistema de arquivos criptografado, o que incluiu a verificação e otimização da implementação da biblioteca. No gráfico é possível ver também que a latência de encriptação é, de maneira geral, menor na WAESlib quando comparado com a Libcrypto, sendo indicada para aplicações que são mais sensíveis ao tempo de resposta e menos à quantidade de dados encriptados. O maior valor de latência apresentado para 8 *threads* clientes se dá pela limitação de recursos do processador, o que faz a função aguardar o processamento do contexto, se refletindo em seu tempo.

Na Figura 4.3 pode-se ver que o número de *threads* trabalhadoras possui grande influência na vazão e na latência da WAESlib. A função WAES_encrypt necessita do contexto pronto para dar prosseguimento na encriptação dos dados. Com poucas *threads* trabalhadoras

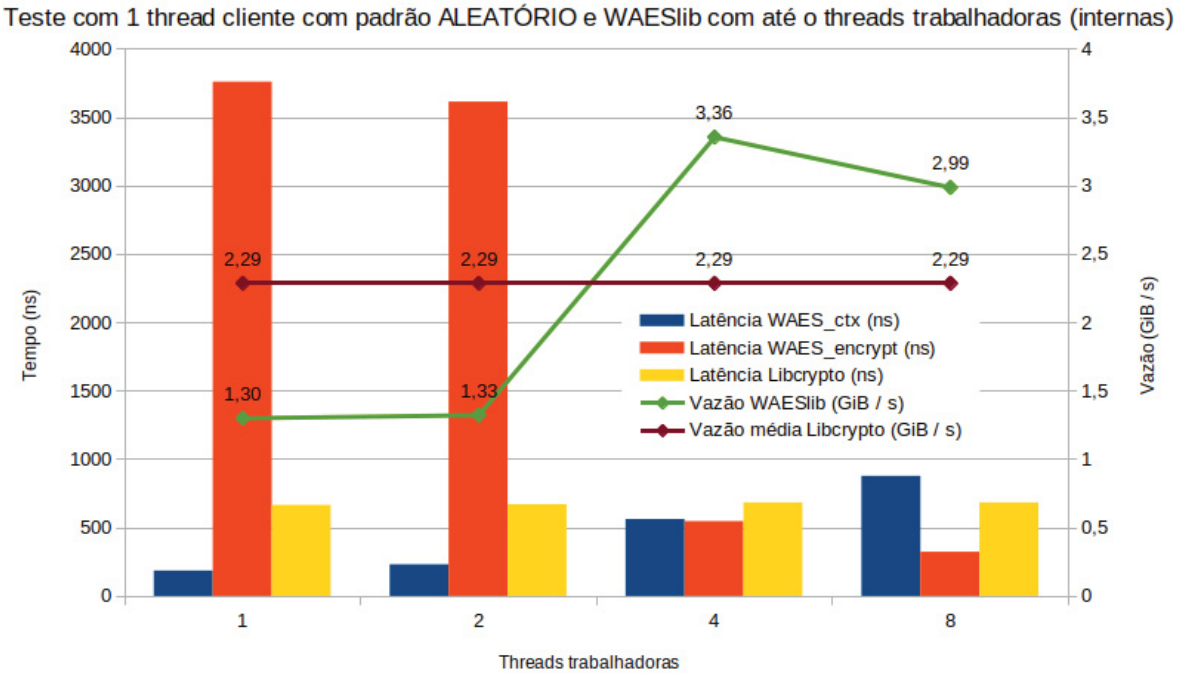


Figura 4.3: Resultado do teste de requisição aleatória variando o número de *threads* trabalhadoras.

muitas vezes o função é alcançada antes do contexto estar pronto o que faz com que ela tenha que aguardar, refletindo na alta latência apresentada. A vazão também é bastante afetada com o número correto de *threads* uma vez que com poucas trabalhadoras elas não possuem poder computacional suficiente para superar as necessidades de trocas de contexto e maximizar a vazão, porém com o número adequado conseguem superar a encriptação utilizando diretamente a Libcrypto, pois realizam processamento pesado da encriptação antes do tempo.

5 IMPLEMENTAÇÃO E TESTES DO SISTEMA DE ARQUIVOS ENCFS++ EM CPU

Durante a segunda etapa do trabalho foi realizada a implementação e otimização do sistema de arquivos EncFS++ em sua versão CPU, fazendo uso da biblioteca WAESlib CPU. Após breves testes de corretude, garantindo o bom funcionamento do sistema de arquivos, foram realizados diversos experimentos e otimizações de modo a conseguir o melhor desempenho. A Seção 5.1 apresenta os ambientes utilizados durante os testes. Na Seção 5.2 são mostrados os resultados dos testes com valores variados de clientes, enquanto na Seção 5.3 é feita uma análise similar, porém variando o a quantidade de *threads* trabalhadoras.

5.1 AMBIENTES DE TESTE

Uma característica importante da versão CPU do EncFS++ é a sua versatilidade, uma vez que pode ser utilizada nos mais diversos equipamentos sem a necessidade de uma hardware específico para o seu funcionamentos. Essa versatilidade traz consigo a necessidade de realização de testes em diversos ambientes de modo a verificar o comportamento e desempenho do sistema nas mais variadas situações. De modo a cobrir uma grande gama de sistemas foram utilizados três parâmetros que acarretam num total de oito ambientes diferentes de testes.

O primeiro parâmetro é o tipo de equipamentos, sendo ele físico ou virtual. Para os testes em ambiente físico foi utilizado um computador com um processador Intel i7-11700F, com 2.50GHz, 8 Cores e 16 threads e 16GB de memória RAM, e para os testes em equipamento virtualizado foi utilizada uma máquina virtual, sobre um virtualizador VMware ESXi 6.7, que possui 8 CPUs virtuais, cada uma com uma thread e ligada a um core do processador Xeon Gold 5218 de 2,3GHz, 16GB de memória RAM.

O segundo parâmetro é relacionado com os dispositivos nos quais o sistema de arquivos estará montado, uma vez que as características dos dispositivos podem alterar o desempenho do sistema de arquivos. Para o equipamento físico foram utilizados dois dispositivos de armazenamento, sendo um SSD Kingston SA400S37 e um NVMe ADATA SX6000LNP, já para o equipamento virtual foram utilizados discos virtuais, porém montados sobre HDDs e SSDs físicos.

O terceiro parâmetro se dá sobre o uso ou não da tecnologia AES-NI. A versatilidade do EncFS++ CPU permite que este seja colocado em equipamentos muito mais simples do que desktops ou servidores e que não possuem aceleração em hardware para criptografia. De modo a simular a operação em equipamentos como esses foram testadas versões com e sem o uso do AES-NI.

De modo a diminuir possíveis divergências de software que pudessem afetar o resultado dos experimentos todos os testes foram realizados em um sistema operacional Ubuntu 18.04, com Openssl versão 1.1.1 e libfuse versão 2.9.7. Além disso o EncFS++ é um sistema de arquivos empilhado, necessitando de um sistema de arquivos base. Para todos os testes foi utilizado como base o sistema ext4.

Para todos os testes foi utilizado programa *Filebench* com as cargas *fileserver.f*. Esta carga simula a utilização de um servidor de arquivos, onde arquivos grandes são criados completos, editados, lidos completos e deletados. Os testes são definidos para serem realizados por um determinado período de tempo, 60 segundos, e não por número de operações. Por esse motivo os testes são realizados com número variável de operações e, como a vazão é calculada de acordo com cada uma delas se fez necessário verificar se o número de operações eram suficientes para

que os resultados fossem significativos. Após os testes foi verificado que o número de operações estava entre 20 e 200 mil, dependendo da vazão alcançada no ambiente de teste, não sendo necessária a realização de outros testes, uma vez que a vazão é calculada como a média das operações. A quantidade de operações realizadas por teste, a facilidade de se reproduzir as cargas e a capacidade de se projetar padrões complexos de testes [16] são algumas das vantagens que justificam o uso do *benchmark* sintético. O EncFS++ GPU definiu como 64 contextos o tamanho padrão para sua janela de contextos. Embora adequado para sua versão em GPU testes preliminares com o *Filebench* mostraram que o tamanho de 64 não é o ideal para a versão em CPU, dessa maneira os testes apresentados foram realizados com uma janela de tamanho 8 por ter sido a que apresentou o melhores resultados nos testes preliminares.

5.2 EXPERIMENTOS VARIANDO O NÚMERO DE CLIENTES

Os primeiros testes realizados foram variando o número de clientes realizando requisições ao sistema de arquivos, sendo representados pelo número de *threads* utilizados pelo *Filebench*. Diferente da versão em GPU, na versão em CPU tanto as requisições como as respostas são processadas no mesmo lugar levando o número de clientes a causar um impacto maior no desempenho do sistema. As figuras de 5.1 a 5.4 apresentam os gráficos para os testes nos diversos ambientes, mostrando os resultados tanto para o EncFS++ como para o EncFS.

Para cada um dos gráficos apresentados, o eixo x corresponde ao número de *threads* utilizadas pelo *Filebench*, simulando uma quantidade de clientes. A quantidade de clientes simulados varia entre 1 e 10. Como em todos os ambientes o número de núcleos de processamento é 8 valores muito maiores do que esse acarretam em perda de desempenho. Para o eixo y temos a vazão, dada em MB/s, e o *speedup* do EncFS++ em relação ao EncFS.

A análise dos gráficos mostra que apesar da diferença nos valores absolutos o comportamento do sistema em todos os ambientes de teste se mantém praticamente o mesmo. Para todos os testes a vazão aumenta junto com o número de clientes até chegar ao pico próximo de 7 clientes, ponto após o qual começa a diminuir novamente.

Outro ponto importante é que, sendo iguais os demais parâmetros, o dispositivo de armazenamento parece ter pouco impacto no desempenho do sistema como um todo. De maneira contrária, a vazão aparece ser bastante afetada quando se passa de ambientes físicos para virtualizados, com quedas próximas a 50%, no entanto os sistemas virtualizados se mostram mais resilientes quanto a não utilização de AES-NI, com queda de desempenho próximo de 5% contra 15% do ambiente físico.

Independente do ambiente, no entanto, é possível ver ganhos de desempenho consideráveis quando comparados com o EncFS. Quando comparados os valores máximos de vazão para os dois sistemas vemos que para ambientes com AES-NI temos ganhos de mais de 170% em todos os equipamentos e dispositivos. Para ambientes sem AES-NI no entanto é possível ver uma diferença grande no *speedup* quando comparados os ambientes virtualizados e os físicos. Enquanto nos ambientes físicos são encontrados aumentos da vazão em mais de 320% nos ambientes virtuais essa porcentagem sobe para mais de 560%. Essa diferença se da devido a resiliência do EncFS++ quando executado em sistemas com menor poder computacional. Os valores exatos dos resultados dos testes podem ser encontrados no Anexo A

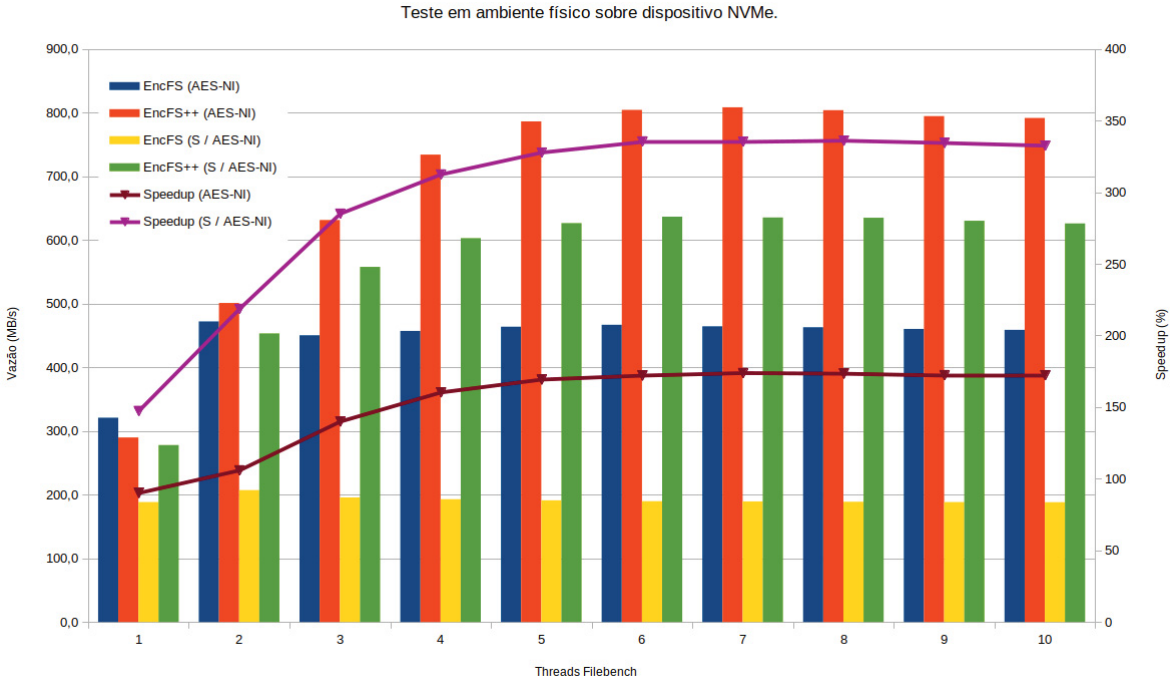


Figura 5.1: Teste variando a quantidade de clientes em ambiente físico sobre dispositivo NVMe.

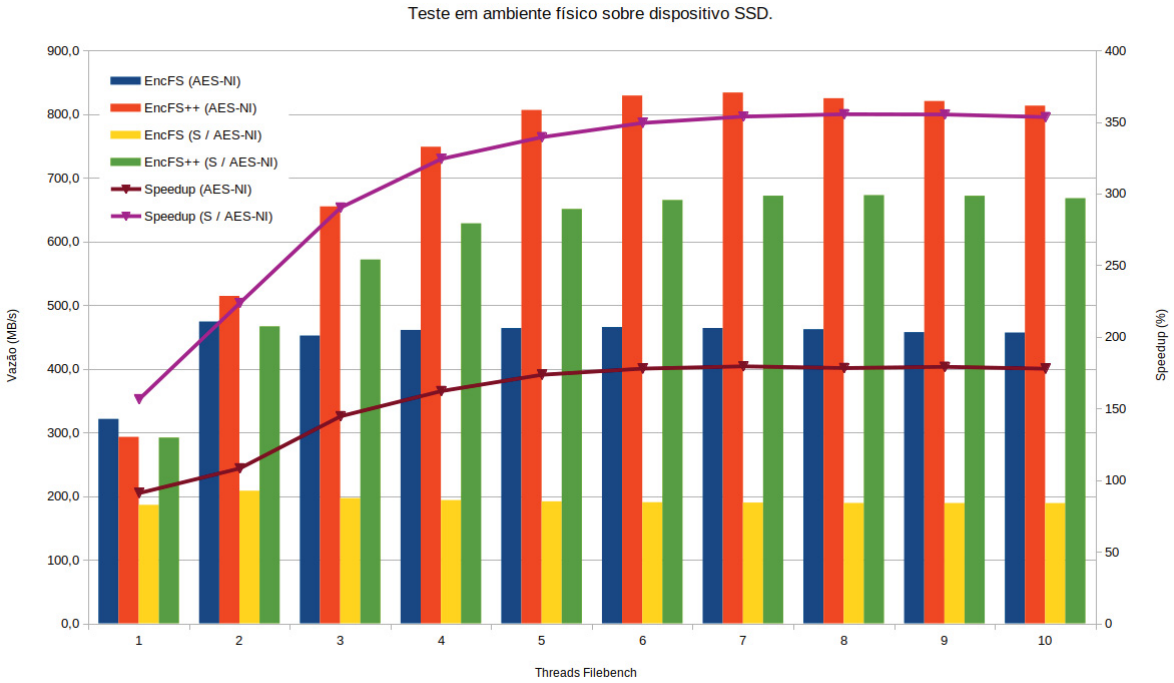


Figura 5.2: Teste variando a quantidade de clientes em ambiente físico sobre dispositivo SSD.

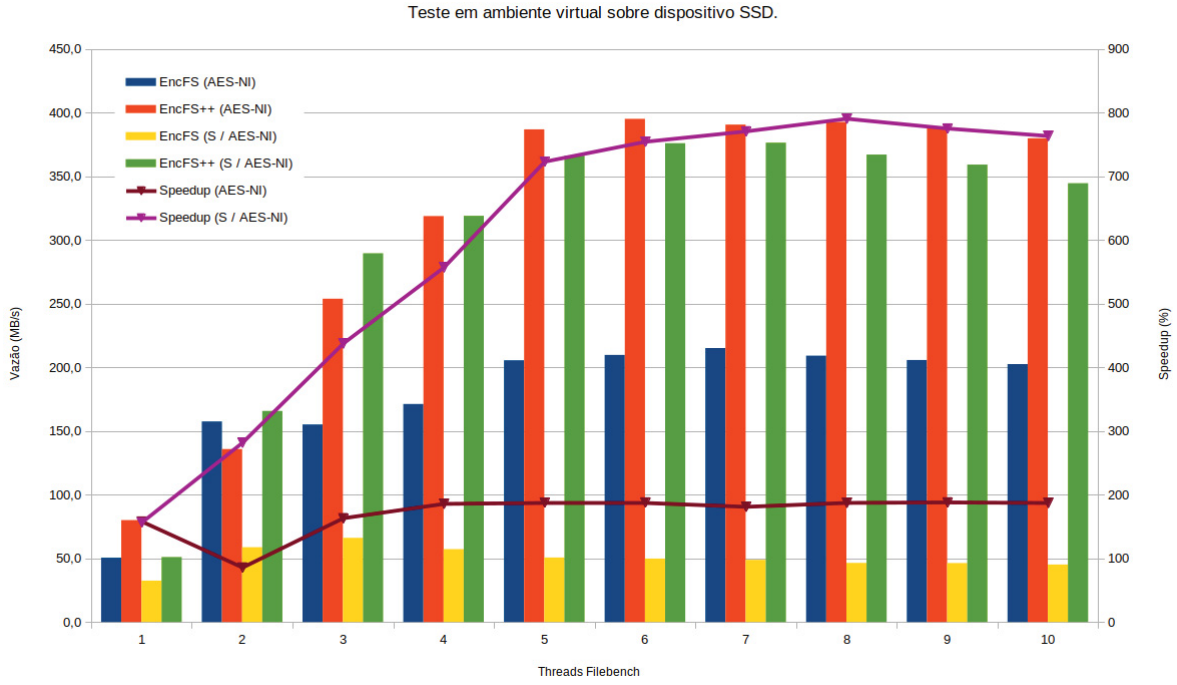


Figura 5.3: Teste variando a quantidade de clientes em ambiente virtual sobre dispositivo SSD.

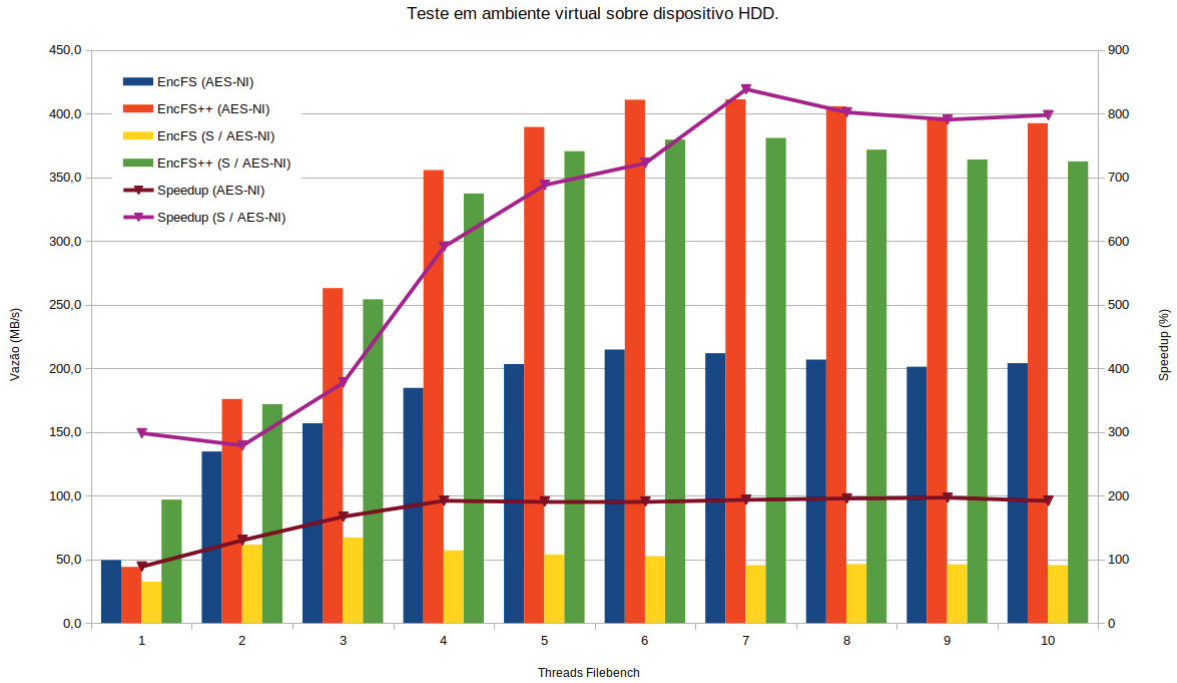


Figura 5.4: Teste variando a quantidade de clientes em ambiente virtual sobre dispositivo HDD.

5.3 EXPERIMENTO VARIANDO O NÚMERO DE THREADS TRABALHADORAS

Os gráficos apresentados nas figuras de 5.5 a 5.12 mostram o resultado dos experimentos alterando o número de *threads* trabalhadoras. Foram testados valores entre 1 e 8 devido ao número de núcleos de processamento. O número de clientes utilizados em cada um dos testes

varia, tendo sido escolhidos os valores para os quais eram encontrados os valores máximos de vazão para cada caso.

Em todos os testes é possível ver uma perda de desempenho quando se aumenta o número de *threads* trabalhadoras. uma vez que o sistema é processado em CPU é necessário realizar o gerenciamento das *threads* e dos dados de modo a não permitir o acesso simultâneo a um mesmo contexto por partes diferentes da aplicação. O processo de gerenciamento consome tempo e recursos e acaba se tornando custoso com um número grande de *threads* trabalhadoras, resultando em uma queda no desempenho.

Outro ponto importante a ser notado é que o custo atrelado ao gerenciamento das *threads* afeta de maneira mais significativa sistemas com mais poder de processamento. A WAESlib, e como consequência o EncFS++, faz uso de duas técnicas buscando melhorar o desempenho do processo sendo eles a encriptação especulativa e o processamento paralelo. Enquanto a encriptação especulativa apresenta melhorias em todos os ambientes testados, verificado quando comparado com os valores encontra pelo sistema EncFS, o processamento paralelo não.

Quando são comparados os ambientes que utilizam AES-NI com os que não utilizam é possível ver claramente uma tendência no comportamento do EncFS++ no que diz respeito ao processamento paralelo. Em todos os ambientes que fazem uso do AES-NI o pico de vazão é encontrado quando se tem apenas uma *thread* trabalhadora, indicando que as perdas com gerenciamento são maiores que os ganhos trazidos por mais trabalhadores. Essa situação acontece pois, como o sistema possui um poder de processamento alto para criptografia o tempo de processamento de um contexto se torna comparável com o tempo utilizado para o gerenciamento. Em contrapartida, em ambientes sem o AES-NI como o processamento dos contextos se torna mais demorado existe um ganho em colocar trabalhadores em paralelo, mesmo com o custo extra atrelado. Essa troca de poder de processamento por processamento paralelo funciona como um amortecedor e diminuindo a queda de desempenho quando comparados dois ambientes. É possível ver esse efeito de amortecimento quando são comparados os ambientes com e sem AES-NI para os testes utilizando o EncFS++ e o EncFS. Enquanto a maior queda de desempenho do EncFS++ foi de menos de 20% para ambientes equivalentes com e sem AES-NI a menor queda do EncFS foi maior que 55%.

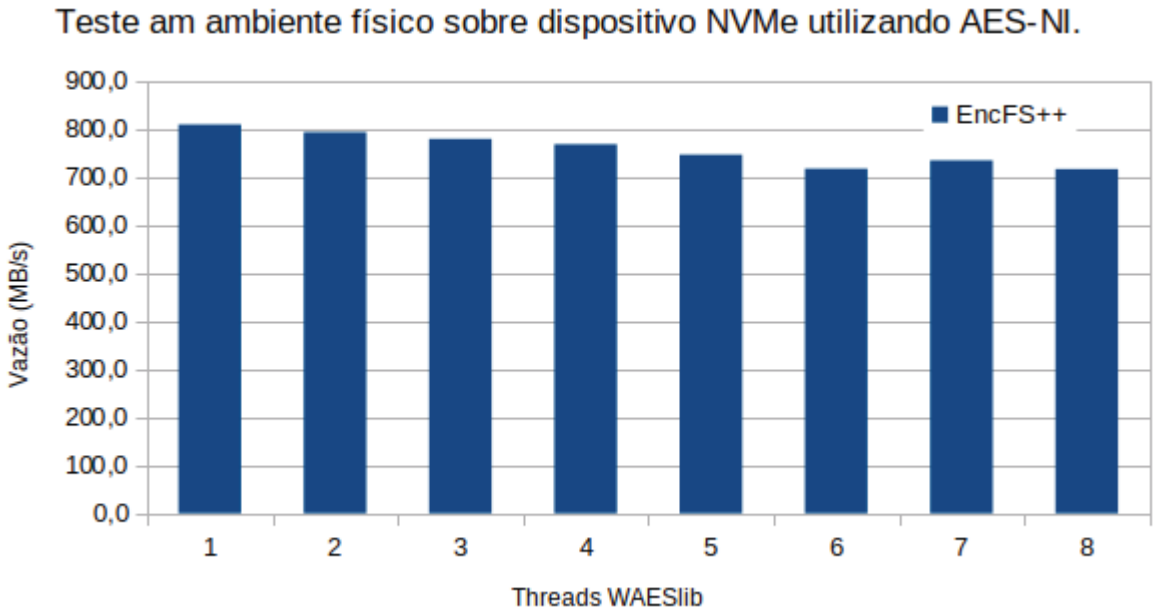


Figura 5.5: Teste variando a quantidade de threads trabalhadoras em ambiente físico sobre dispositivo NVMe e utilizando AES-NI.

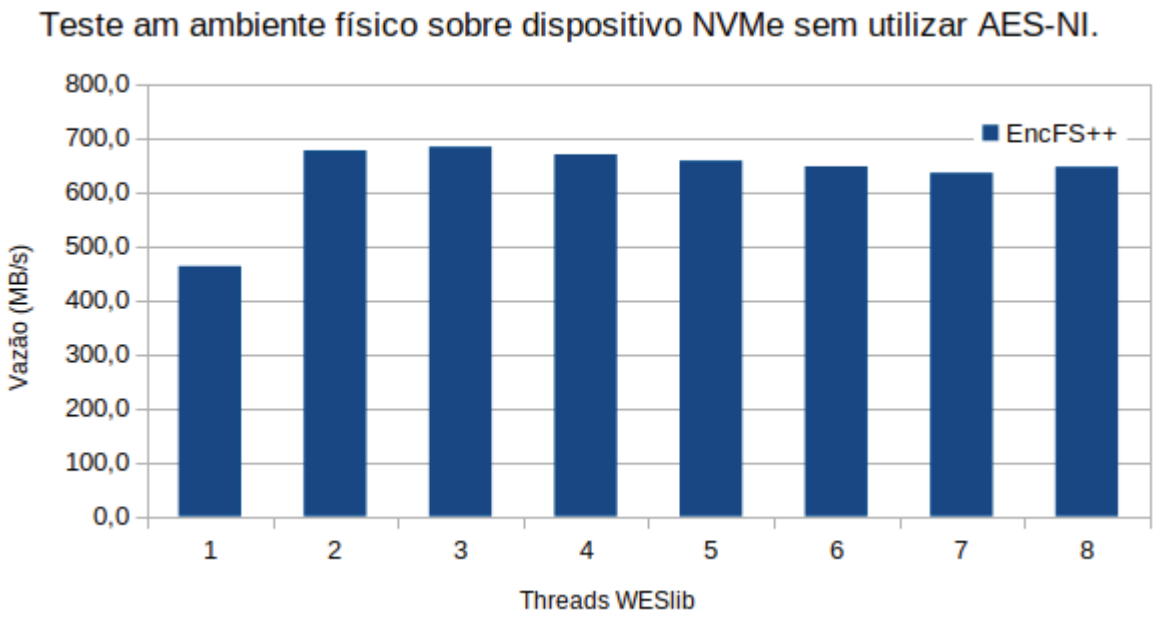


Figura 5.6: Teste variando a quantidade de threads trabalhadoras em ambiente físico sobre dispositivo NVMe e sem utilizar AES-NI.

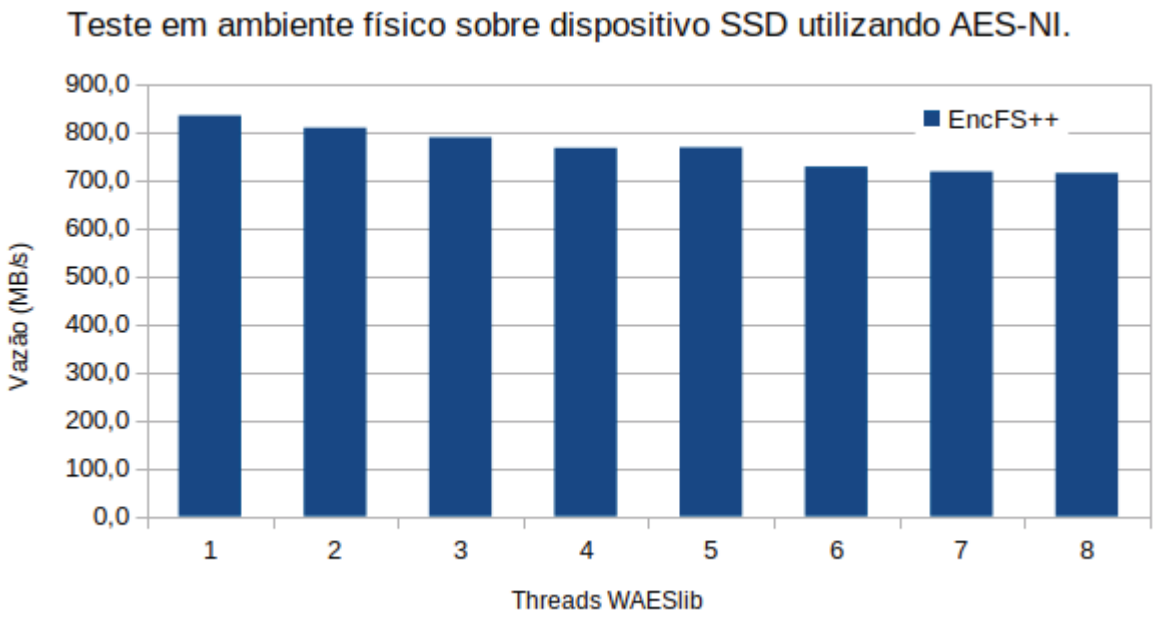


Figura 5.7: Teste variando a quantidade de threads trabalhadoras em ambiente físico sobre dispositivo SSD e utilizando AES-NI.

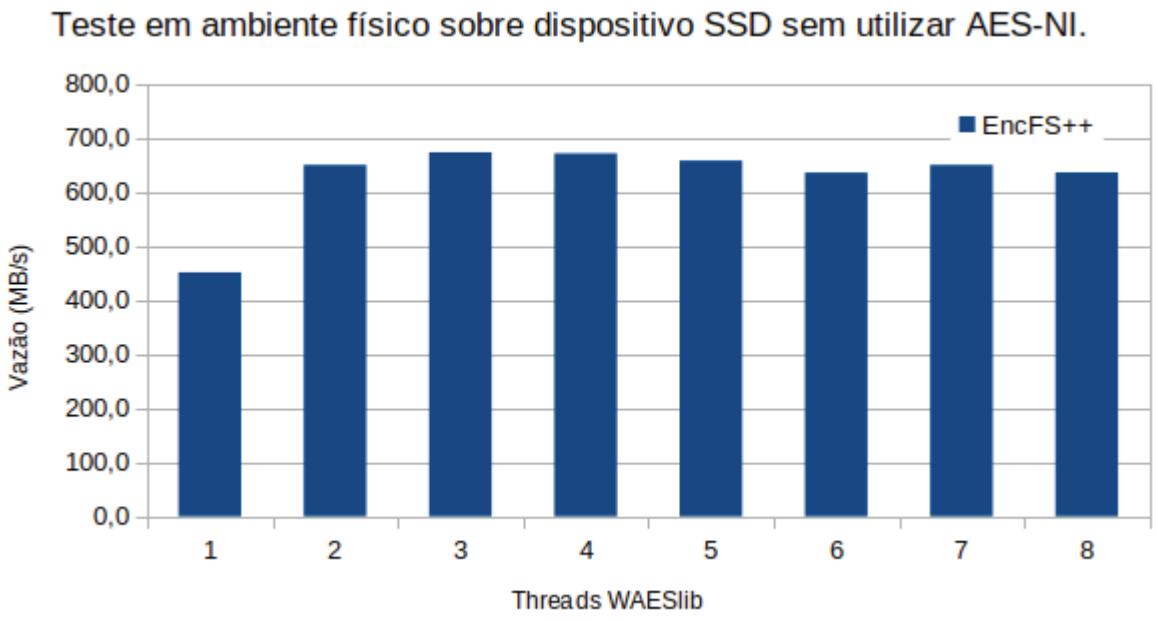


Figura 5.8: Teste variando a quantidade de threads trabalhadoras em ambiente físico sobre dispositivo SSD e sem utilizar AES-NI.

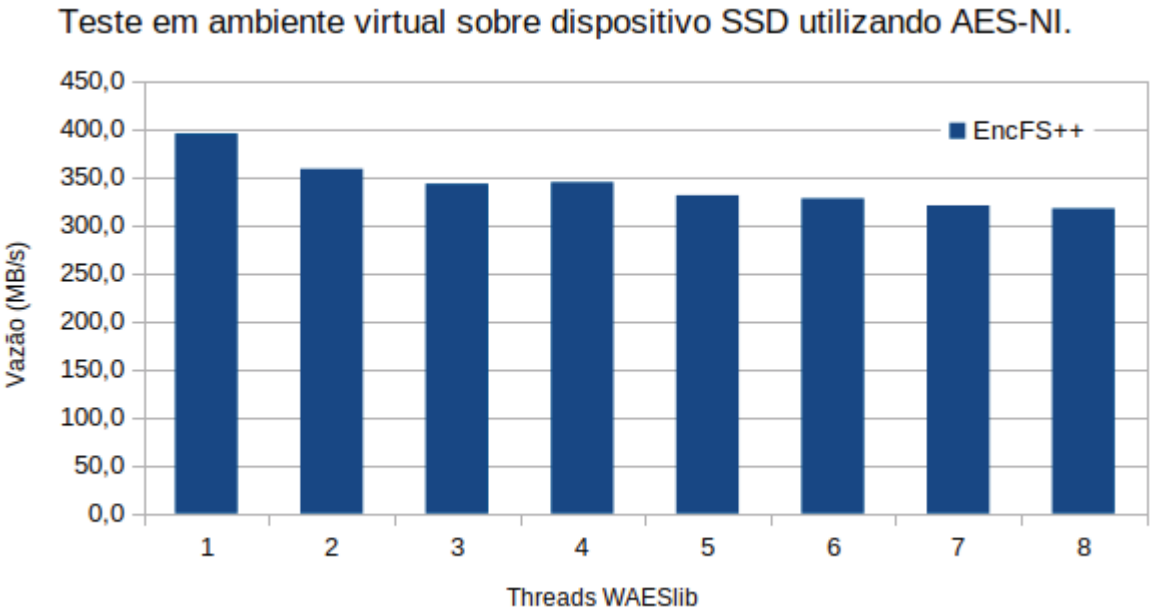


Figura 5.9: Teste variando a quantidade de threads trabalhadoras em ambiente virtual sobre dispositivo SSD e utilizando AES-NI.

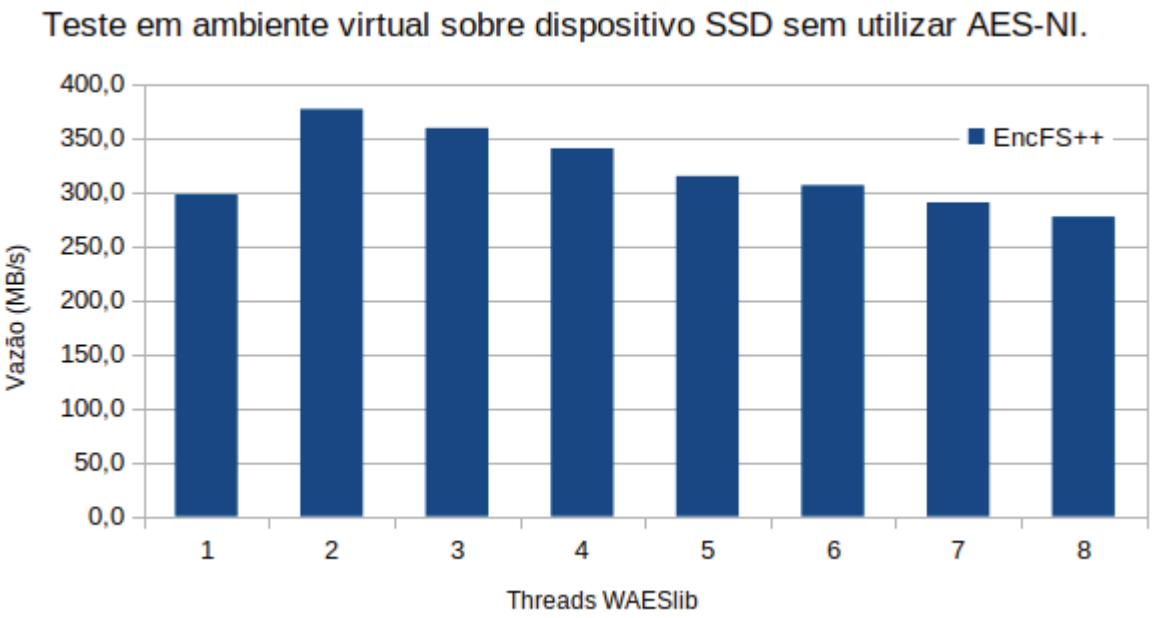


Figura 5.10: Teste variando a quantidade de threads trabalhadoras em ambiente virtual sobre dispositivo SSD e sem utilizar AES-NI.

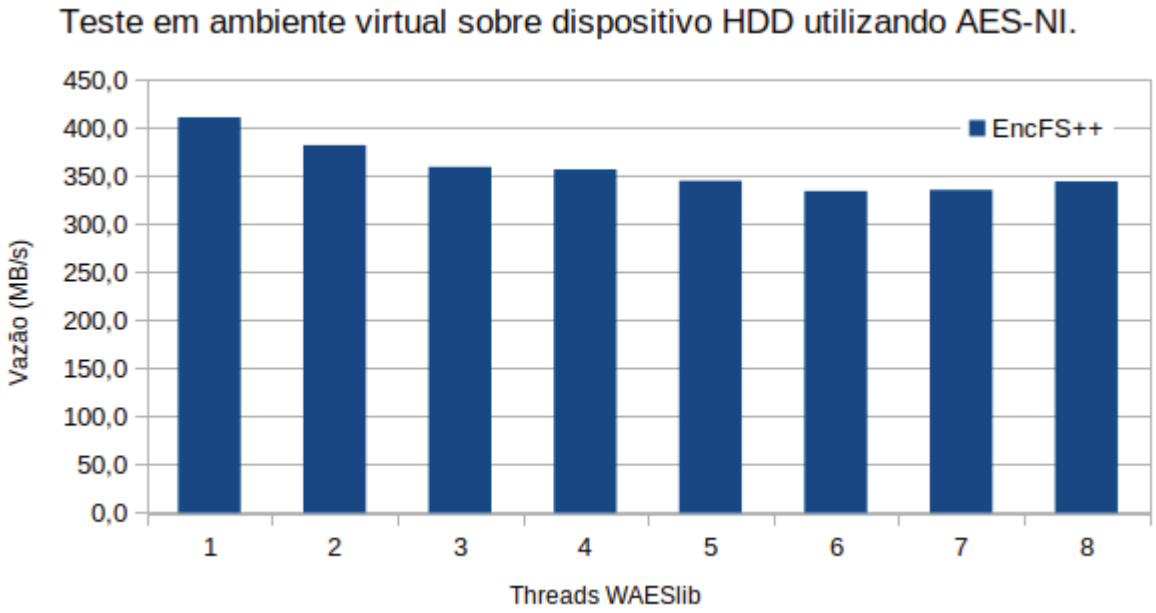


Figura 5.11: Teste variando a quantidade de threads trabalhadoras em ambiente virtual sobre dispositivo HDD e utilizando AES-NI.

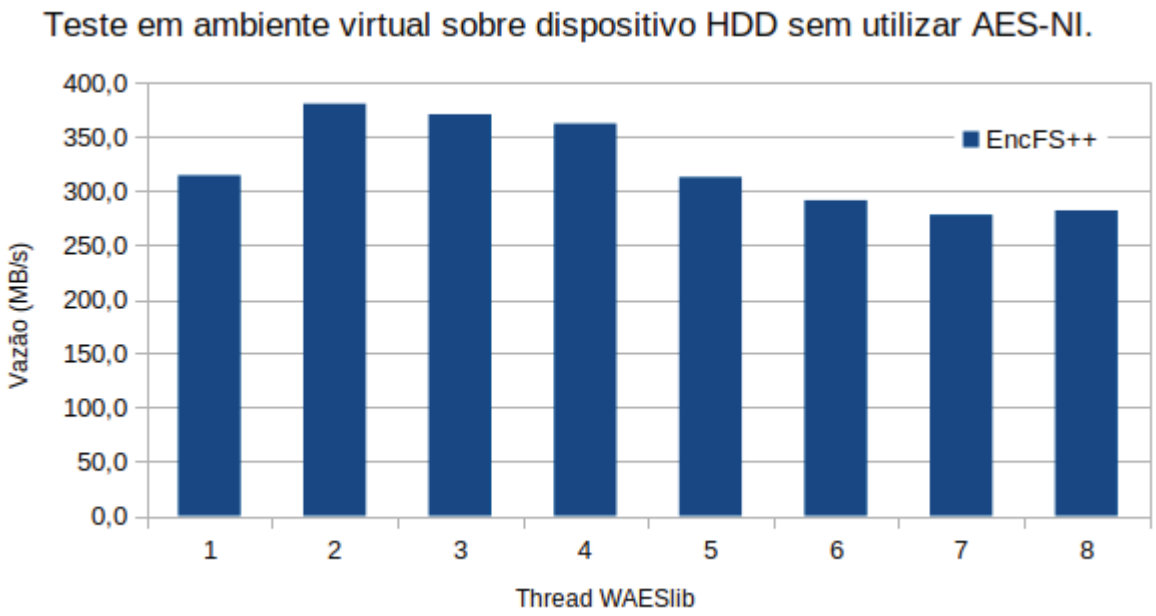


Figura 5.12: Teste variando a quantidade de threads trabalhadoras em ambiente virtual sobre dispositivo HDD e sem utilizar AES-NI.

6 CONCLUSÃO

As vantagens apresentadas pelos modo de operação CTR permitiram a criação e exploração de novas técnicas que buscam a melhoria de desempenho do processo de encriptação e decriptação. A capacidade de pré processamento dos contextos permitiu a criação de técnicas de encriptação especulativa e, por ser completamente paralelizável, muitas destas técnicas focam no utilização GPUs. Buscando fugir da tendência encontrada este trabalho busca explorar as vantagens apresentadas pelo modo CTR utilizando encriptação especulativa em CPUs multicore.

A primeira etapa do trabalho teve como foco e implementação e testes da biblioteca WAESlib em sua versão CPU, uma vez que esta foi criada utilizando CUDA para ser processada em GPUs. Para a utilização do processamento paralelo em CPU foram utilizadas *threads* e de modo a garantir a não interferência de uma *thread* nos dados sendo processados por outra foram utilizados *locks*, o que gerou uma preocupação sobre possíveis perdas de desempenho. Ao final da primeira etapa haviam sido implementadas as funções definidas pela biblioteca de modo que seu funcionamento ocorresse de maneira correta, segura e eficiente com o uso de *threads*, completando seu objetivo.

A segunda etapa do trabalho focou na implementação, testes e calibração do sistema de arquivos EncFS++ em suas versão CPU. Inicialmente foi utilizada a biblioteca WAESlib em sua versão CPU, implementada na primeira etapa, para a implementação do sistema de arquivos e verificado o correto funcionamento do mesmo, não focando em desempenho. Uma vez garantido o correto funcionamento do sistema foram realizados testes e calibrados parâmetros de modo a se obter a mais eficiência possível do sistema. Esta etapa teve como objetivo validar a possibilidade e eficiência da utilização das técnicas WAES em sistemas de arquivos criptografadoos utilizando processamento paralelo em CPUs multicore.

6.1 RESULTADOS OBTIDOS

Os resultados encontrados demonstram a viabilidade da utilização da técnica WAES, bem como a implementação de um sistema de arquivos eficiente que faz uso da técnica, com processamento em CPUs multicore. Não apenas mostrando a possibilidade e o correto funcionamento mas também apresentando consideráveis ganhos de desempenho.

Foi implementada e testada a biblioteca WAESAlib versão em CPU. Durante os testes foi verificado o comportamento esperado para a biblioteca bem como ganhos diretos quando comparados com a Libcrypto, seja em vazão ou em latência.

A implementação do sistema de arquivos EncFS++ utilizando a WAESlib versão CPU apresentou ganhos de vazão bastante elevados quando comparados aos sistemas não preditivos. Em ambientes com AES-NI os ganhos foram não menores que 170%. Para os ambientes sem AES-NI houve aumento de mais de 320% para ambientes físicos e 560% para ambientes virtuais.

Foi estudado o comportamento do sistema realizando variações no número de clientes. Variando o número de clientes foi observada vazão máxima quando a quantidade está próxima ao número de núcleos do processador. Para valores menores a diminuição da vazão indica subutilização do sistema, enquanto que para valores maiores é um indicativo de sobrecarga do processador.

Testes realizados variando o número de *threads* trabalhadoras mostraram a diferença do funcionamento ideal do sistema de arquivos em ambientes diferentes. Enquanto para ambientes com maior poder de processamento os melhores resultados apareceram com apenas uma *thread*,

sem paralelismo, para ambientes menos potentes os melhores resultados são atingidos utilizando paralelismo. Essa característica gera uma resiliência no sistema, evitando grandes perdas de vazão em ambientes similares compensando a perda de poder de processamento com a adição de mais unidades de processamento. Essa resiliência se mostra particularmente relevante em ambientes virtualizados, onde as perdas de vazão foram de menos de 8% quando comparados ambientes similares com e sem AES-NI.

Os resultados obtidos dos diversos testes mostra que o sistema possui comportamento similar em todos os ambientes testados. A indiferença quanto a natureza do ambiente no qual o sistema está sendo utilizado corrobora a validação do sistema bem como sua versatilidade, podendo ser utilizado nos mais variados ambientes sem comportamentos inesperados.

6.2 TRABALHOS FUTUROS

Existem diversas áreas nas quais é possível estender e aprimorar este trabalho, destacando-se dentre elas as áreas de segurança da informação, sistemas operacionais e análise de desempenho.

Um tópico importante é a análise de vulnerabilidades na implementação da WAESlib e da EncFS++ em CPU. A não checagem de possíveis alterações nos contadores, bem como o armazenamento dos contextos pré processados na memória RAM podem apresentar pontos de vulnerabilidade no sistema.

Outro ponto seria a busca da utilização da encriptação especulativa no modo GMC. Além de trazer mais segurança a utilização da encriptação especulativa no modo GMC poderia trazer ganhos de desempenho, mesmo que não seja possível a aplicação direta como é feito com o modo CTR.

Também é possível ampliar a implementação da WAESlib e a criação de um sistema de arquivos para que funcionem diretamente em modo núcleo, diminuindo as perdas geradas pela utilização de um sistema de arquivos em espaço de usuário.

Outra ampliação possível para a WAESlib, porém em outra direção, é a possibilidade de expansão da biblioteca para que possa funcionar simultaneamente tando em CPU quanto em GPU, se adaptando conforme a necessidade do programador.

Para análise de desempenho um estudo interessante seria a implementação do sistema de arquivos EncFS++ de forma dinâmica, adaptando e alterando a quantidade de *threads* trabalhadoras de acordo com o ambiente em que está sendo utilizado.

REFERÊNCIAS

[1] Fernando Antonio Mota Trinta and Rodrigo Cavalcanti de Macêdo. Um estudo sobre criptografia e assinatura digital. https://www.cin.ufpe.br/~flash/ais98/cripto/criptografia.htm, 1998. Acessado em 18/09/2020.

[2] William Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson, 2017.

[3] Specification for the advanced encryption standard (aes). http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001. Acessado em 18/09/2020.

[4] Contribuidores da Wikipédia. Block cipher mode of operation. https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation, 2020. Acessado em 18/09/2020.

[5] Andrews Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, 2015.

[6] Kate Stewart et al. 2020 linux kernel history report. Technical report, Linux Foundation, 2020.

[7] Werner Fischer. Linux storage stack diagram. https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram, 2018. Acessado em 26/09/2020.

[8] Mohammad Banikazemi, David Daly, and Bulent Abali. Sysman: A virtual file system for managing clusters. In *Proceedings of the 22nd Large Installation System Administration Conference (LISA '08)*, San Diego, CA, USA, November 2008.

[9] Chris Lomont. Introduction to intel advanced vector extensions. June 2011.

[10] Kashish Goyal and Supriya Kinger. Modified caesar cipher for better security enhancement. *International Journal of Computer Applications*, 73:26–31, 07 2013.

[11] John Rydning David Reinsel, John Gantz. The digitization of the world - from edge to core. Technical Report US44413318, Seagate, 2018.

[12] National Institute of Standards and Technology. Cryptographic key management workshop summary – june 8-9, 2009. https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7609.pdf, 2010. Acessado em 18/09/2020.

[13] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[14] Vandeir Eduardo. Sistema de arquivos criptográfico com aceleração especulativa em GPU. Master's thesis, Pós-Graduação em Informática - Universidade Federal do Paraná., Curitiba - PR, 2018.

[15] Wagner M. Nunan Zola and L. C. E. de Bona. Parallel speculative encryption of multiple aes contexts on GPUs. *2012 Innovative Parallel Computing (InPar)*, pages 1–9, 2012.

[16] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[17] Data encryption standard (des). https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf, 1999. Acessado em 18/09/2020.

[18] Network Working Group. The esp triple des transform. https://tools.ietf.org/html/rfc1851, 1995. Acessado em 18/09/2020.

[19] Mihir Bellare and Phillip Rogaway. *Introduction to Modern Cryptography*. University of California at Davis, 2005.

[20] Recommendation for block cipher modes of operation. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf, 2001. Acessado em 20/09/2020.

[21] Christof Paar and Jan Pelzl. *Understanding Cryptography*. Springer, 2010.

[22] Niels Ferguson, Bruce Scheneier, and Tadayoshi Kohno. *Criptography Engineering*. Wiley, 2010.

[23] Robert Love. *Linux Kernel Development*. Developer's Library, 2010.

[24] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2005.

[25] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To fuse or not to fuse: Performance of user-space file systems. In *Proceedings of the 15th USENIX Conference onFile and Storage Technologies (FAST '17)*, Santa Clara, CA, USA, March 2017.

[26] Internet Engineering Task Force (IETF). Network file system (nfs) version 4 protocol. https://tools.ietf.org/html/rfc7530, 2015. Acessado em 26/09/2020.

[27] Contribuidores do projeto. encfs. https://github.com/vgough/encfs, 2020. Acessado em 26/09/2020.

[28] Tyler Hicks, Dustin Kirkland, and Michael Halcrow. ecryptfs. https://www.ecryptfs.org/, 2020. Acessado em 02/01/2021.

[29] Milan Broz. Dmcrypt. https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt, 2020. Acessado em 02/01/2021.

[30] Oracle Corporation and/or its affiliates. What is zfs? https://docs.oracle.com/cd/E19253-01/819-5461/zfsover-2/, 2010. Acessado em 02/01/2021.

[31] Open ZFS. Openzfs documentation. https://openzfs.github.io/openzfs-docs/, 2020. Acessado em 02/01/2021.

[32] The Linux Kernel. Filesystem-level encryption (fscrypt). https://www.kernel.org/doc/html/latest/filesystems/fscrypt.html, 2020. Acessado em 02/01/2021.

[33] Tecnologia de extensões do conjunto de instruções intel. https://www.intel.com.br/content/www/br/pt/support/articles/000005779/processors.html, 2020. Acessado em 26/09/2020.

[34] Intel advanced vector extensions 512 (intel avx-512). https://www.intel.com.br/content/www/br/pt/architecture-and-technology/avx-512-animation.html, 2020. Acessado em 26/09/2020.

[35] Shay Gueron. Intel advanced encryption standard (aes) new instructions set. Technical Report 323641-001, Intel, 2010.

[36] Jens Axboe. 1. fio - flexible i/o tester rev. 3.25. https://fio.readthedocs.io/en/latest/fio_doc.html, 2017. Acessado em 02/01/2021.

[37] William Norcott. Iozone filesystem benchmark. http://www.iozone.org/, 2016. Acessado em 02/01/2021.

[38] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *login Usenix Mag.*, 41, 2016.

[39] 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, 2012.

[40] Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. Fast software aes encryption. FSE'10, page 75–93, Berlin, Heidelberg, 2010. Springer-Verlag.

[41] 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.

[42] Guang liang Guo, Quan Qian, and Rui Zhang. Different implementations of aes cryptographic algorithm. *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 1848–1853, 2015.

[43] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. pages 206–213, 01 2010.

[44] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. Terra incognita: On the practicality of user-space file systems. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, Santa Clara, CA, July 2015. USENIX Association.

[45] Bharath Kumar Reddy Vangoor, Prafful Agarwal, Manu Mathew, Arun Ramachandran, Swaminathan Sivaraman, Vasily Tarasov, and Erez Zadok. Performance and resource utilization of fuse user-space file systems. *ACM Trans. Storage*, 15(2), May 2019.

[46] Angelos D. Keromytis, Columbia University, and Jason L. Wright. Filebench. The Design of the OpenBSD Cryptographic Framework, 2003.

[47] Chien-Kai Tseng, Shang-Chieh Lin, and Y. Hsu. A file system using GPU-accelerated file-wise reliability scheme. In *Proceedings of the International Conference on Parallel and Distributed Processing Tech- niques and Applications (PDPTA 12)*, page 32–38, Las Vegas, 2012. CSREA Press.

[48] Shang-Chieh Lin, Yu-Cheng Liao, and Yarsun Hsu. A reliable and secure GPU-assisted file system. *Algorithms and Architectures for Parallel Processing, Lecture Notes in Computer Science*, 8630:71–84, 08 2014.

[49] Vandeir Eduardo, Luis C. Erpen de Bona, and Wagner M. Nunan Zola. Speculative encryption on GPU applied to cryptographic file systems. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 93–105, Boston, MA, February 2019. USENIX Association.

APÊNDICE A – RESULTADOS E GRÁFICOS DOS TESTES DO ENCFS++

Este apêndice contém as tabelas e gráficos completos referentes aos testes apresentados no Capítulo 5. Cada seção a seguir representa um ambiente diferente onde os testes foram realizados.

A.1 SISTEMA FÍSICO COM DISPOSITIVO NVME

A.1.1 Resultados utilizando AES-NI

Tabela A.1: Resultados dos testes de vazão (MB/s) realizados em ambiente físico sobre NVMe utilizando AES-NI.

EncFS++										
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Threads WAESlib	**1**	289,7	501,1	631,3	734,0	786,3	804,3	808,4	803,9	794,5	791,6
	2	315,0	522,1	653,4	739,0	775,2	792,7	792,4	785,6	780,2	777,8
	3	304,1	516,6	647,8	734,9	763,8	776,8	778,5	774,7	770,2	764,1
	4	302,9	511,0	639,0	720,1	748,2	759,0	767,5	759,0	753,8	747,7
	5	299,8	506,6	625,3	699,6	733,8	743,8	746,0	738,8	733,5	732,3
	6	291,3	491,8	600,5	669,9	705,9	714,0	716,7	705,9	704,8	708,4
	7	295,1	503,6	613,8	619,9	727,4	733,3	734,1	726,6	722,5	640,1
	8	292,6	493,9	597,7	679,4	708,9	719,7	715,9	707,7	706,5	704,5
EncFS											
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
		320,8	472,1	450,3	457,2	463,8	466,8	464,4	463,0	460,3	458,8

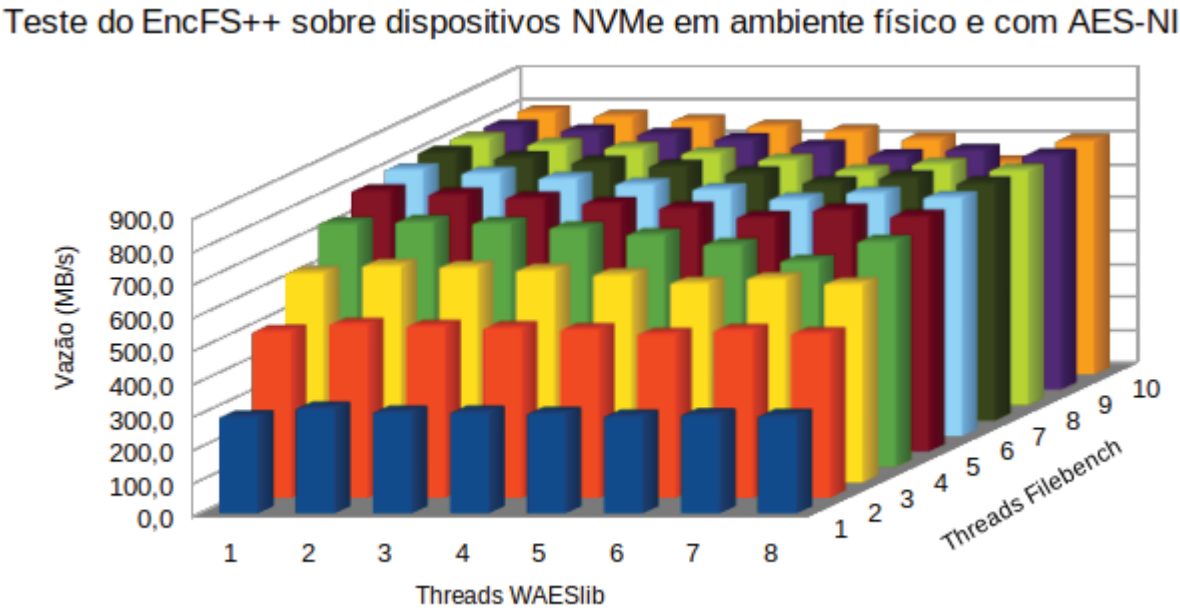


Figura A.1: Comportamento dos testes realizados sobre o EncFS++ em ambiente físico sobre dispositivo NVMe e utilizando AES-NI.

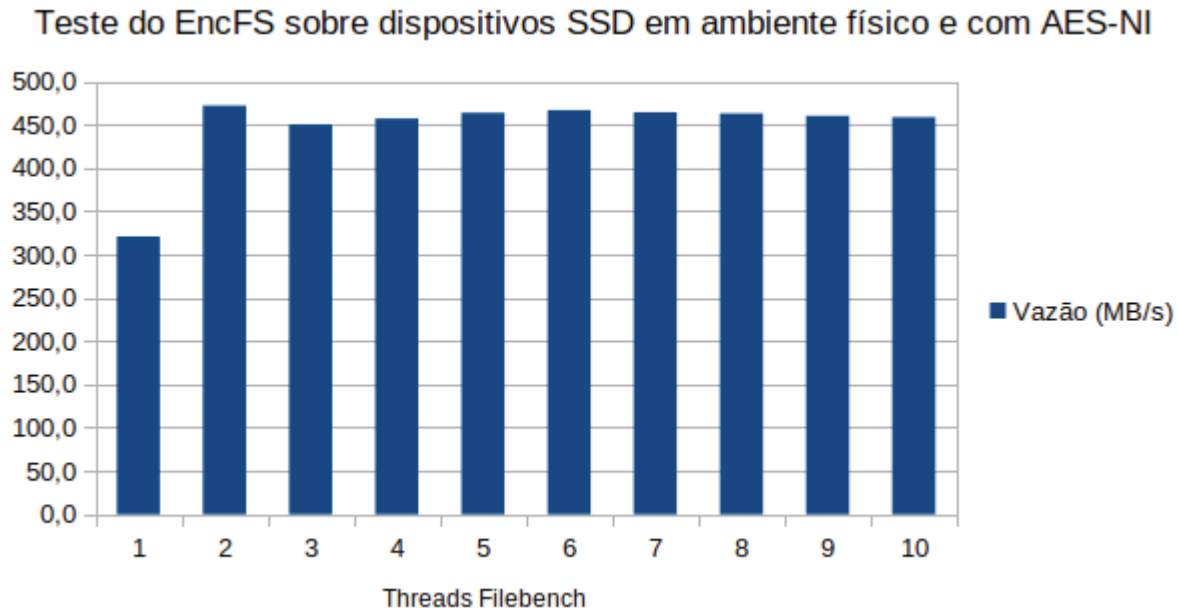


Figura A.2: Comportamento dos testes realizados sobre o EncFS em ambiente físico sobre dispositivo NVMe e utilizando AES-NI.

A.1.2 Resultados não utilizando AES-NI

Tabela A.2: Resultados dos testes de vazão (MB/s) realizados em ambiente físico sobre NVMe sem utilizar AES-NI.

EncFS++										
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Threads WAESlib	**1**	247,4	413,2	465,9	469,5	464,9	462,5	458,0	447,4	447,0	440,6
	2	272,2	450,2	558,3	625,0	664,5	676,1	668,5	660,8	649,4	640,9
	3	291,7	466,5	571,8	634,8	665,1	682,9	682,8	676,2	673,5	671,3
	4	288,4	467,3	572,5	622,0	653,6	669,0	670,2	664,6	659,3	655,8
	5	284,8	466,0	571,1	616,8	645,3	657,4	657,5	653,3	651,0	645,7
	6	281,6	464,1	567,7	610,9	635,3	646,9	648,3	642,5	639,0	636,8
	7	279,7	460,9	554,9	601,5	629,6	635,0	638,2	633,2	633,3	632,6
	8	283,7	470,7	567,7	615,8	643,6	646,0	647,1	643,4	638,2	632,5
EncFS										
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
		188,3	207,1	195,5	192,8	191,0	189,6	189,3	188,8	188,3	188,1

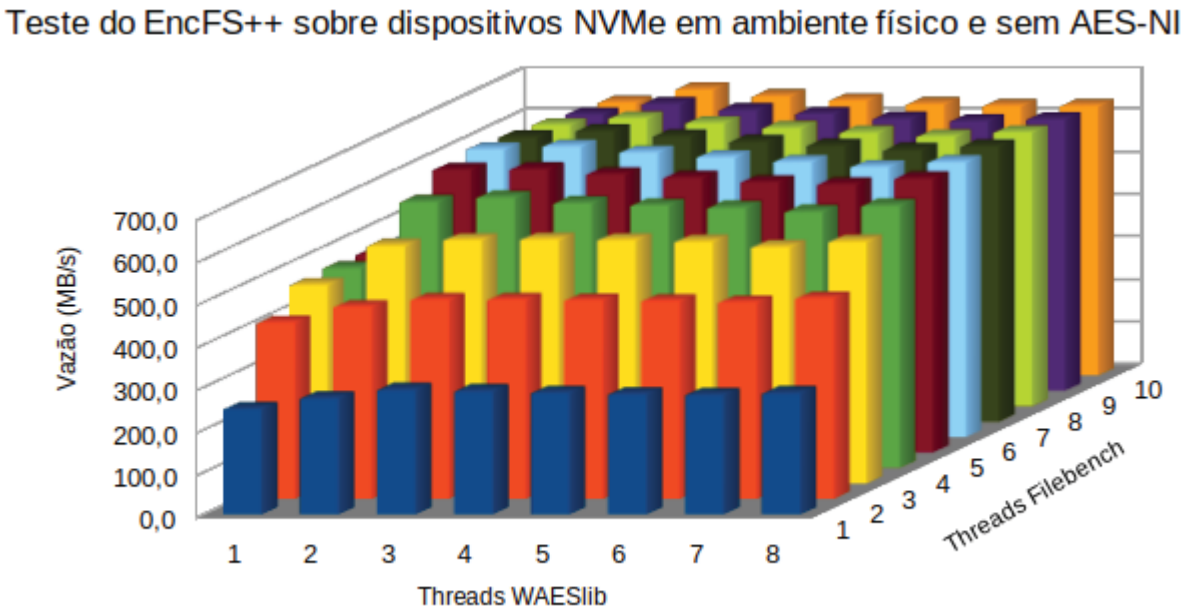


Figura A.3: Comportamento dos testes realizados sobre o EncFS++ em ambiente físico sobre dispositivo NVMe e sem utilizar AES-NI.

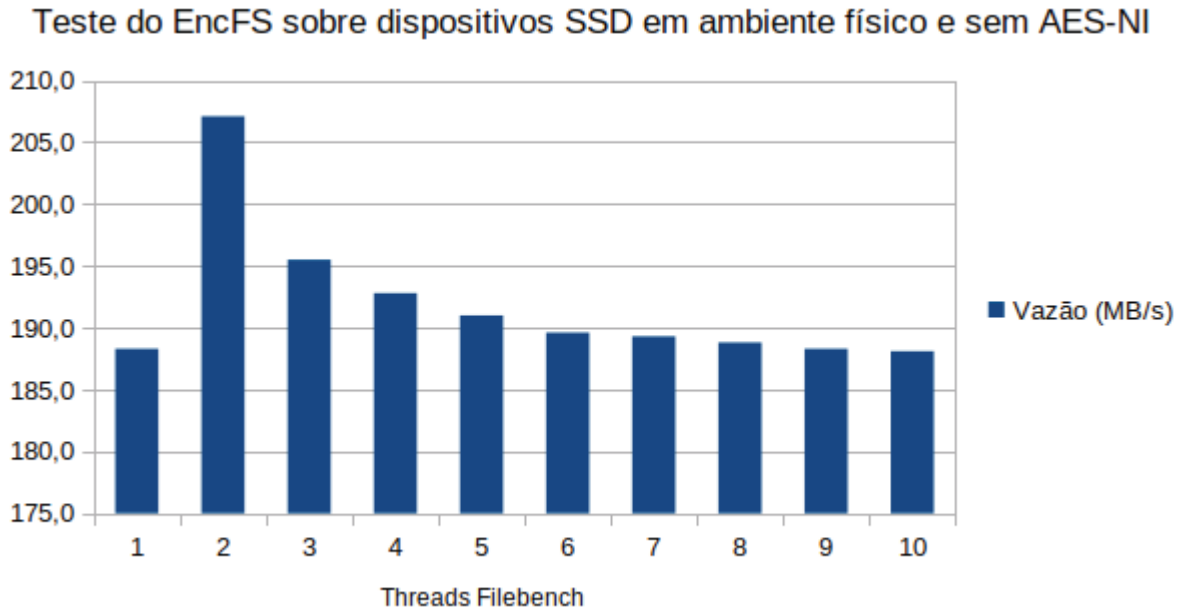


Figura A.4: Comportamento dos testes realizados sobre o EncFS em ambiente físico sobre dispositivo NVMe e sem utilizar AES-NI.

A.2 SISTEMA FÍSICO COM DISPOSITIVO SSD

A.2.1 Resultados utilizando AES-NI

Tabela A.3: Resultados dos testes de vazão (MB/s) realizados em ambiente físico sobre SSD utilizando AES-NI.

EncFS++										
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Threads WAESlib	**1**	292,8	514,3	654,9	748,6	806,3	829,0	833,7	824,8	820,4	813,2
	2	315,2	528,8	653,5	751,2	788,7	801,9	808,0	800,9	793,8	785,7
	3	304,7	517,8	649,3	738,3	771,3	786,7	788,0	780,7	776,5	772,3
	4	301,2	513,6	641,9	726,0	755,1	770,6	766,2	771,3	765,1	754,8
	5	297,0	514,1	638,8	719,5	759,6	762,3	767,2	762,8	750,8	750,2
	6	295,4	496,3	613,7	690,9	715,6	730,4	727,2	726,9	724,0	718,5
	7	292,1	497,0	601,5	663,6	705,1	715,9	716,8	712,1	708,8	707,2
	8	290,5	488,3	593,0	658,2	701,0	713,2	713,8	699,1	705,9	702,1
EncFS											
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
		321,0	474,1	452,1	460,9	463,8	465,4	463,9	462,0	457,4	456,7

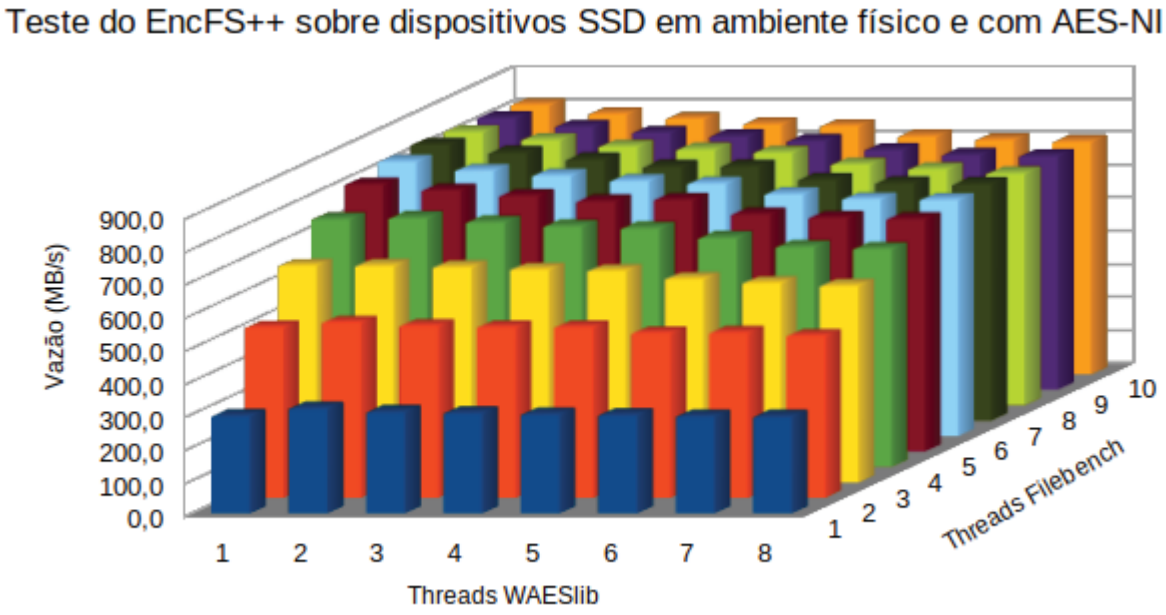


Figura A.5: Comportamento dos testes realizados sobre o EncFS++ em ambiente físico sobre dispositivo SSD e utilizando AES-NI.

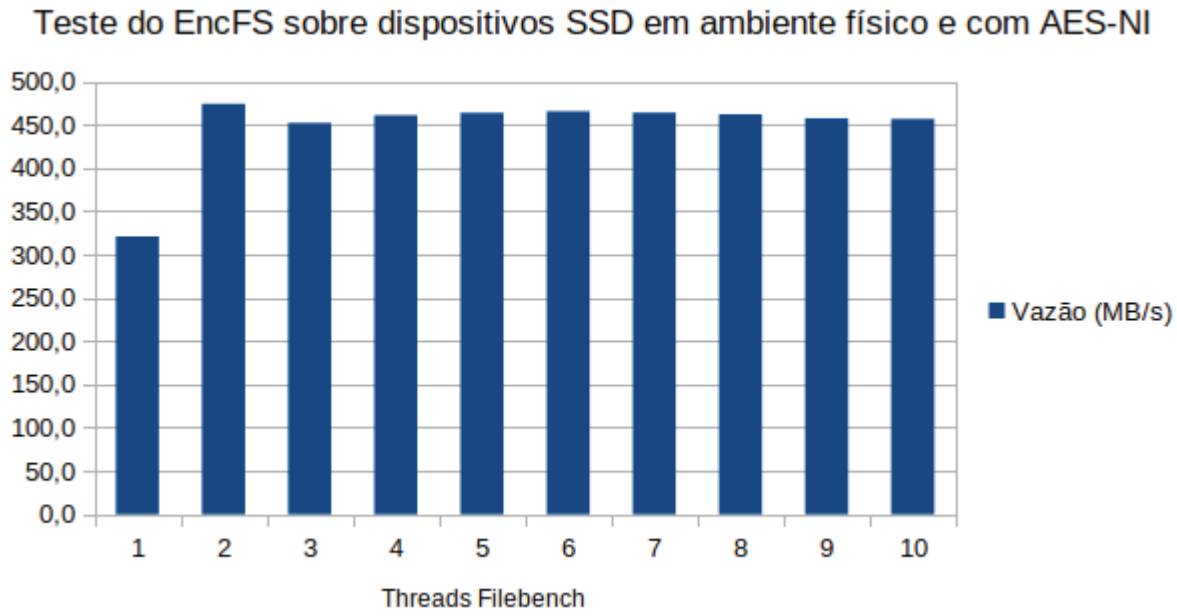


Figura A.6: Comportamento dos testes realizados sobre o EncFS em ambiente físico sobre dispositivo SSD e utilizando AES-NI.

A.2.2 Resultados não utilizando AES-NI

Tabela A.4: Resultados dos testes de vazão (MB/s) realizados em ambiente físico sobre SSD sem utilizar AES-NI.

EncFS++											
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Threads WAESlib	**1**	237,3	405,9	457,7	462,1	456,9	456,8	446,9	450,9	445,5	450,9
	2	272,9	447,7	554,6	614,0	652,6	658,6	666,7	649,5	646,1	640,6
	3	291,7	466,5	571,5	628,3	650,9	665,0	671,7	672,7	671,6	667,8
	4	289,4	468,0	574,0	624,0	655,3	669,4	669,0	670,6	655,8	658,7
	5	283,7	461,5	568,7	618,2	645,9	662,0	661,0	657,3	655,3	649,7
	6	281,3	465,3	563,1	609,0	634,7	646,7	641,1	635,3	635,8	640,9
	7	285,8	465,7	572,4	619,5	646,9	651,0	653,3	649,5	644,2	643,0
	8	279,5	460,2	551,6	596,4	629,1	641,3	636,3	635,6	633,0	641,3
EncFS											
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
		186,1	208,3	196,7	193,6	191,6	190,2	189,7	189,1	188,9	188,8

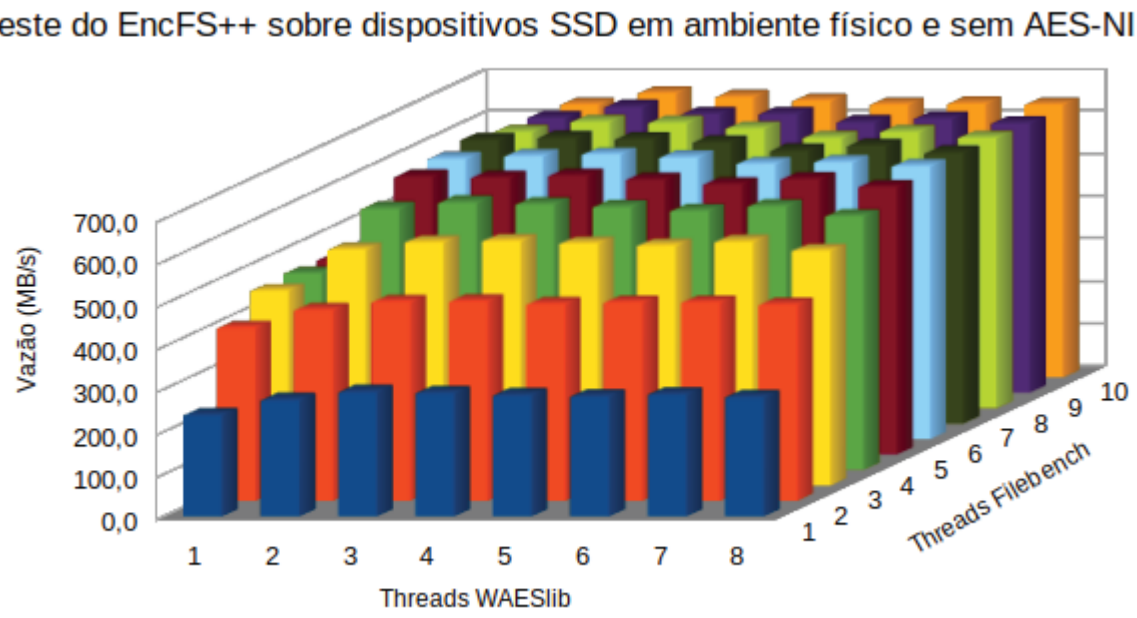


Figura A.7: Comportamento dos testes realizados sobre o EncFS++ em ambiente físico sobre dispositivo SSD e sem utilizar AES-NI.

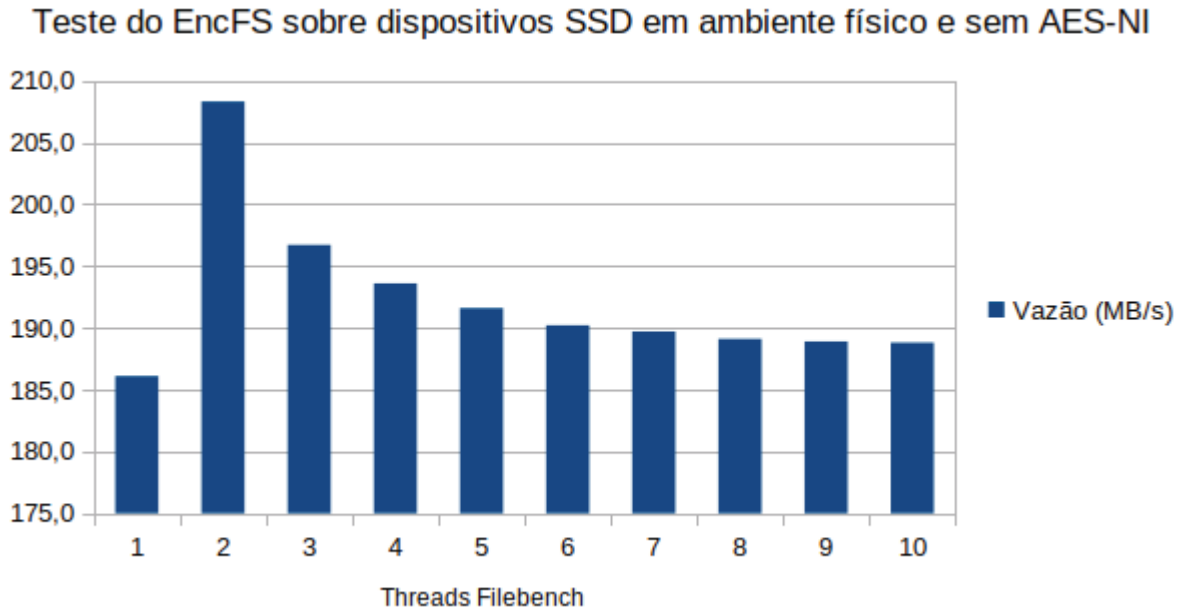


Figura A.8: Comportamento dos testes realizados sobre o EncFS em ambiente físico sobre dispositivo SSD e sem utilizar AES-NI.

A.3 SISTEMA VIRTUAL COM DISPOSITIVO SSD

A.3.1 Resultados utilizando AES-NI

Tabela A.5: Resultados dos testes de vazão (MB/s) realizados em ambiente virtual sobre SSD utilizando AES-NI.

EncFS++										
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Threads WAESlib	**1**	80,0	135,8	253,9	318,8	386,9	395,1	390,6	392,7	387,7	379,9
	2	51,7	145,1	273,4	319,4	337,8	358,3	366,6	361,9	362,1	360,0
	3	84,5	178,7	253,1	301,4	328,7	342,9	355,7	346,5	339,0	340,1
	4	49,9	147,6	257,4	307,3	327,6	344,5	341,0	332,5	336,8	323,2
	5	48,2	233,8	239,4	303,3	311,9	330,8	330,5	328,5	331,8	321,6
	6	47,7	184,8	227,7	267,8	313,7	327,6	327,3	317,7	320,6	313,8
	7	58,8	181,1	252,0	279,9	310,2	320,3	344,2	323,1	327,2	326,1
	8	58,0	182,3	249,8	293,1	327,7	317,2	321,5	320,8	309,3	319,4
EncFS										
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
		50,5	157,5	155,1	171,2	205,5	209,7	215,1	209,1	205,7	202,5

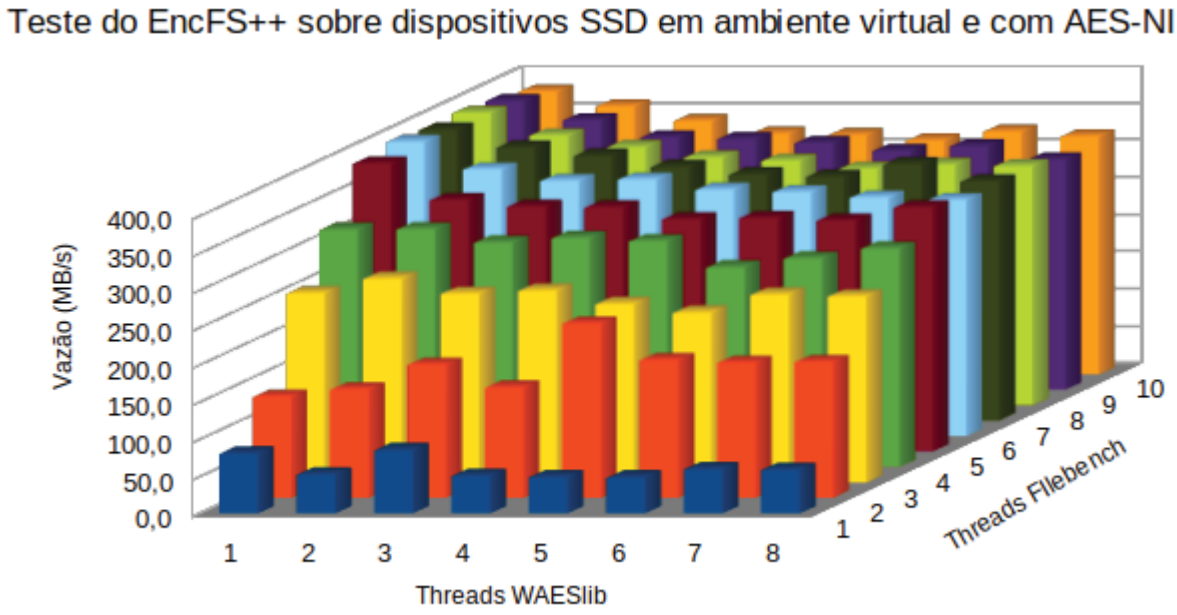


Figura A.9: Comportamento dos testes realizados sobre o EncFS++ em ambiente virtual sobre dispositivo SSD e utilizando AES-NI.

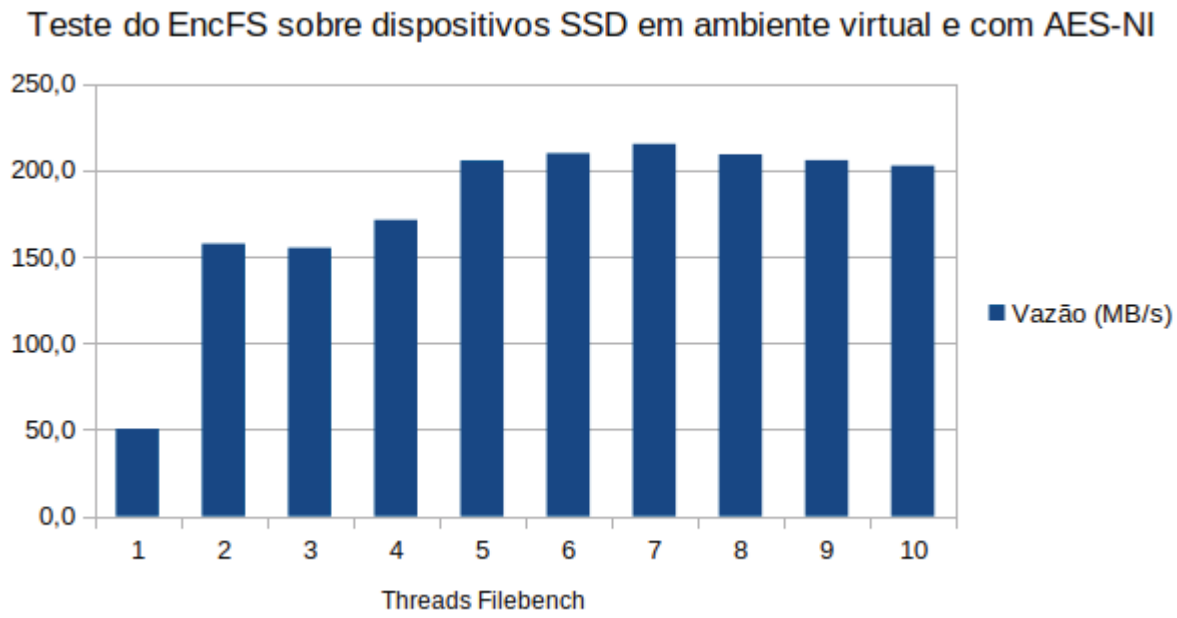


Figura A.10: Comportamento dos testes realizados sobre o EncFS em ambiente virtual sobre dispositivo SSD e utilizando AES-NI.

A.3.2 Resultados não utilizando AES-NI

Tabela A.6: Resultados dos testes de vazão (MB/s) realizados em ambiente virtual sobre SSD sem utilizar AES-NI.

EncFS++										
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Threads WAESlib	**1**	63,4	158,5	235,0	289,0	295,1	295,9	297,4	300,1	303,5	302,4
	2	51,0	165,7	289,6	319,0	366,2	376,0	376,4	367,1	359,2	344,6
	3	74,9	182,5	271,4	320,2	351,8	361,4	358,8	355,7	349,1	332,8
	4	64,6	163,1	263,3	305,2	344,9	344,5	340,1	333,2	332,5	321,3
	5	10,7	207,2	269,1	295,0	308,0	321,4	314,5	310,2	304,3	303,9
	6	89,5	173,1	223,0	268,7	296,3	303,7	306,1	307,4	292,9	295,3
	7	62,4	138,2	241,1	262,2	274,7	285,7	290,2	284,6	295,2	289,8
	8	41,0	158,2	216,1	260,5	268,2	288,5	276,9	291,8	278,9	288,0
EncFS										
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
		32,5	58,7	66,1	57,2	50,6	49,8	48,8	46,4	46,3	45,1

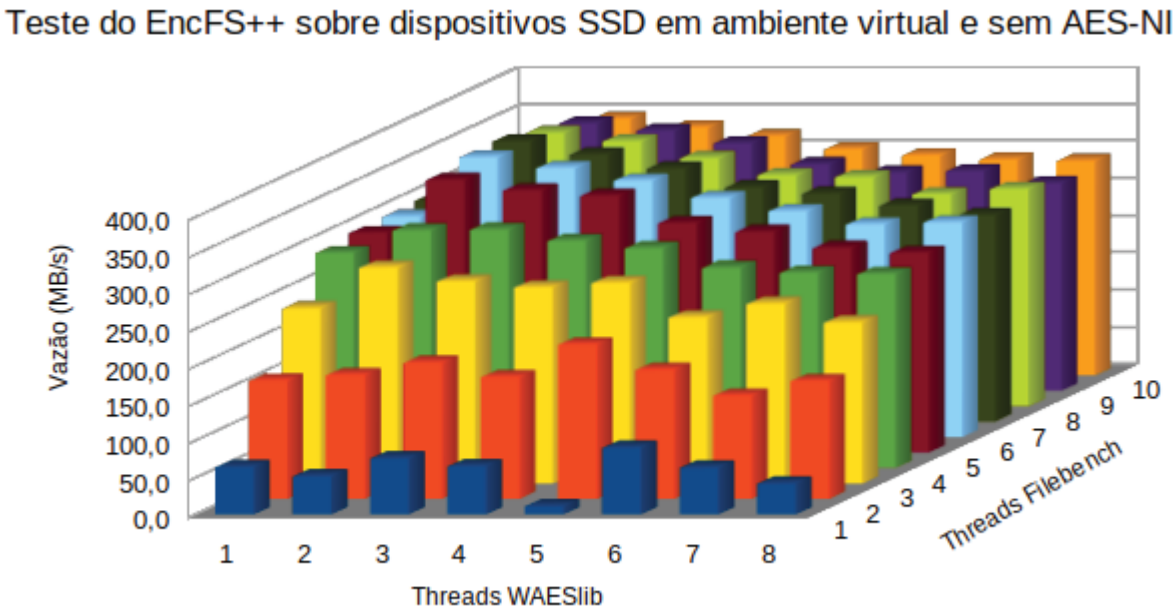


Figura A.11: Comportamento dos testes realizados sobre o EncFS++ em ambiente virtual sobre dispositivo SSD e sem utilizar AES-NI.

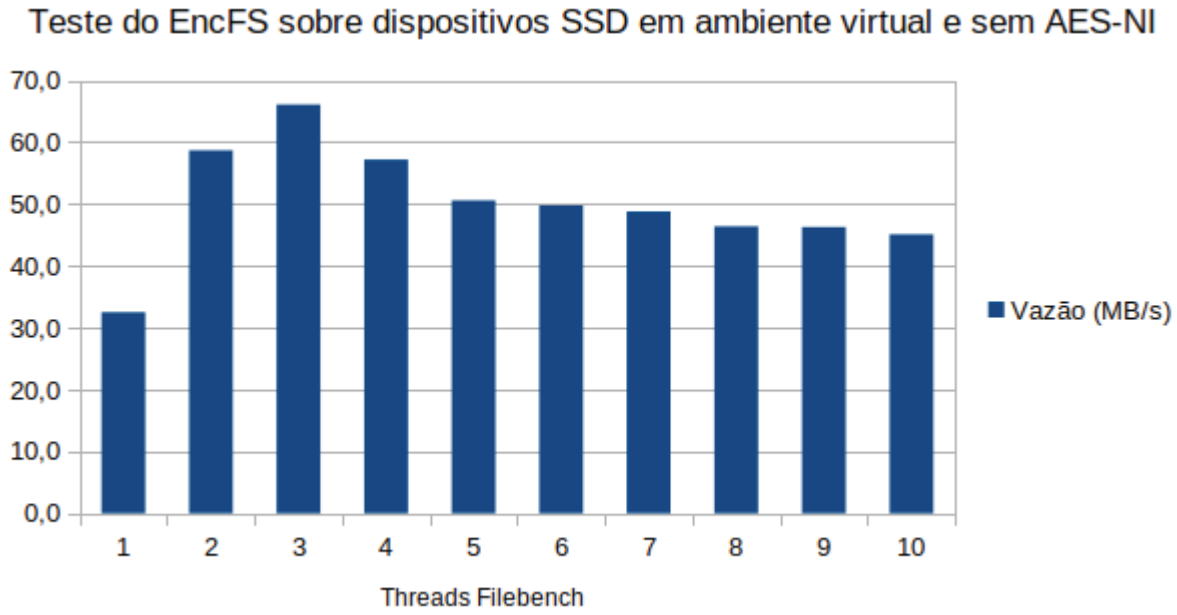


Figura A.12: Comportamento dos testes realizados sobre o EncFS em ambiente virtual sobre dispositivo SSD e sem utilizar AES-NI.

A.4 SISTEMA VIRTUAL COM DISPOSITIVO HDD

A.4.1 Resultados utilizando AES-NI

Tabela A.7: Resultados dos testes de vazão (MB/s) realizados em ambiente virtual sobre HDD utilizando AES-NI.

EncFS++										
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Threads WAESlib	**1**	44,1	175,9	263,0	355,7	389,5	410,9	411,2	405,8	397,4	392,5
	2	54,2	181,9	264,1	302,6	363,7	375,3	382,3	379,0	369,3	368,7
	3	45,2	168,5	252,0	308,9	338,9	356,9	359,6	316,2	350,7	350,2
	4	88,6	146,1	266,1	282,8	334,1	346,9	356,9	341,2	345,0	342,5
	5	55,5	159,5	251,0	293,5	323,4	353,9	345,2	349,8	351,5	339,2
	6	87,5	178,5	231,6	290,5	339,8	317,9	334,5	331,8	335,7	328,8
	7	56,6	177,1	252,9	297,4	319,3	339,3	335,8	327,4	321,5	329,5
	8	55,6	173,4	262,1	292,9	313,2	340,2	344,5	333,4	323,8	322,8
EncFS											
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
		49,4	134,6	156,8	184,6	203,3	214,7	211,8	206,8	201,2	204,0

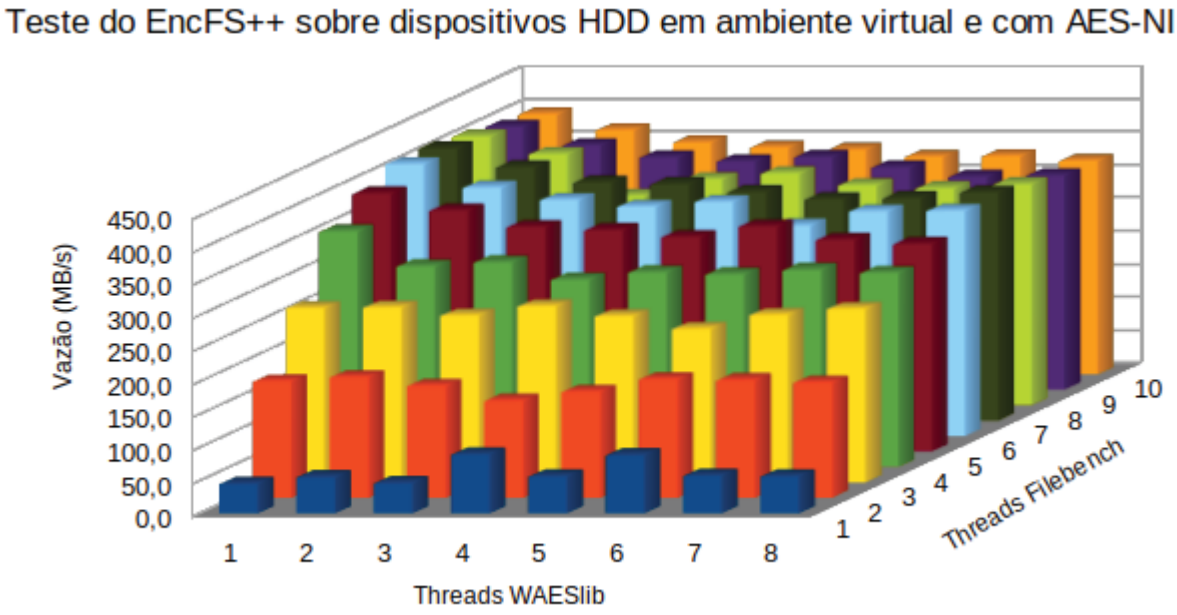


Figura A.13: Comportamento dos testes realizados sobre o EncFS++ em ambiente virtual sobre dispositivo HDD e utilizando AES-NI.

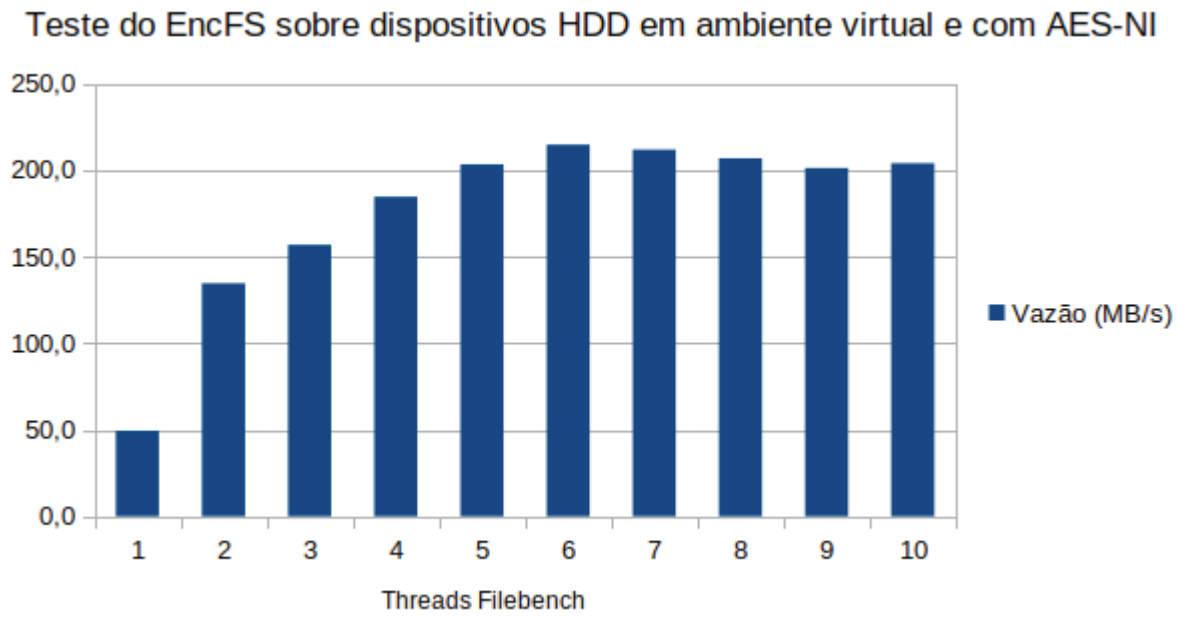


Figura A.14: Comportamento dos testes realizados sobre o EncFS em ambiente virtual sobre dispositivo HDD e utilizando AES-NI.

A.4.2 Resultados não utilizando AES-NI

Tabela A.8: Resultados dos testes de vazão (MB/s) realizados em ambiente virtual sobre HDD sem utilizar AES-NI.

EncFS++										
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Threads WAESlib	**1**	58,7	118,5	225,6	292,9	303,9	309,1	314,8	313,2	314,2	309,3
	2	96,8	171,8	254,2	337,3	370,5	379,6	380,9	371,7	364,0	362,5
	3	96,4	184,2	238,4	318,2	353,1	366,7	371,2	366,8	332,6	350,6
	4	98,7	137,5	222,5	300,4	341,4	359,8	362,7	356,5	326,2	317,1
	5	81,3	161,4	222,7	262,4	300,8	295,4	313,2	310,5	306,4	298,3
	6	81,5	140,7	208,6	244,4	280,9	281,8	291,7	278,4	288,2	285,7
	7	59,8	148,0	210,4	227,0	269,7	273,2	278,4	282,2	275,2	274,4
	8	46,6	133,5	203,7	234,9	272,4	284,7	282,3	270,5	270,0	270,3
EncFS											
Threads Filebench		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
		32,4	61,5	67,2	57,0	53,8	52,5	45,4	46,3	46,0	45,4

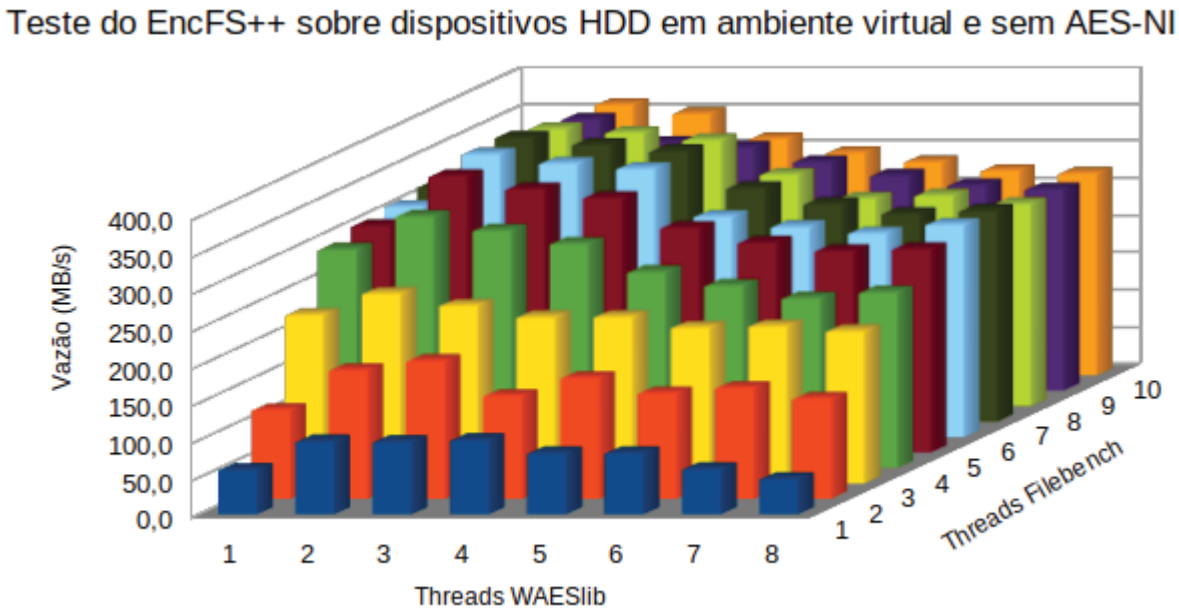


Figura A.15: Comportamento dos testes realizados sobre o EncFS++ em ambiente virtual sobre dispositivo HDD e sem utilizar AES-NI.

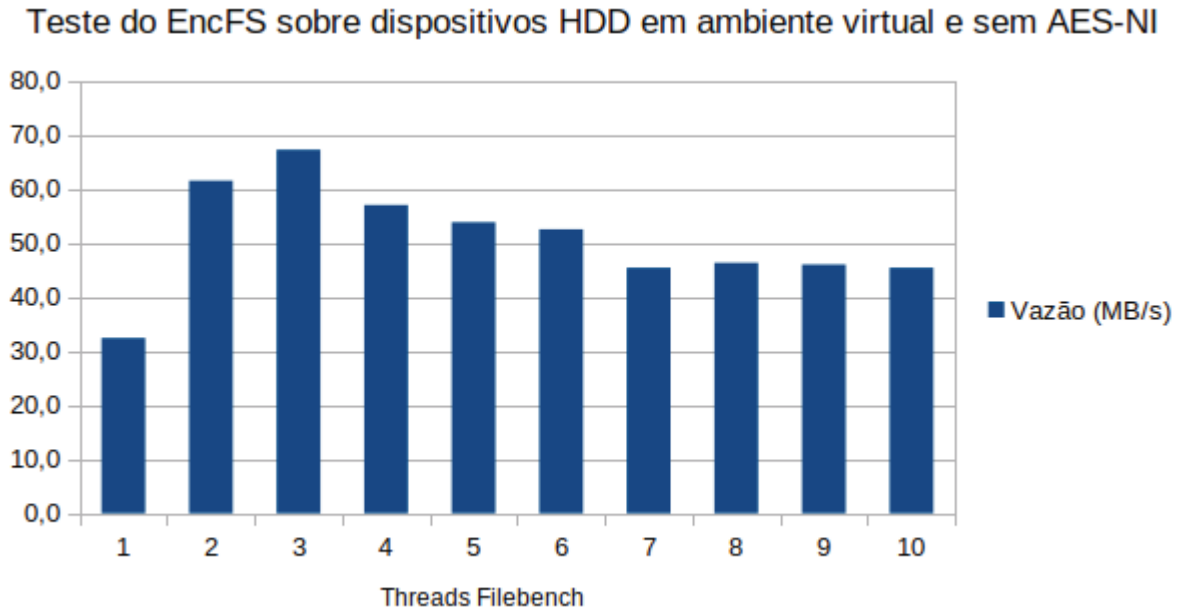


Figura A.16: Comportamento dos testes realizados sobre o EncFS em ambiente virtual sobre dispositivo HDD e sem utilizar AES-NI.