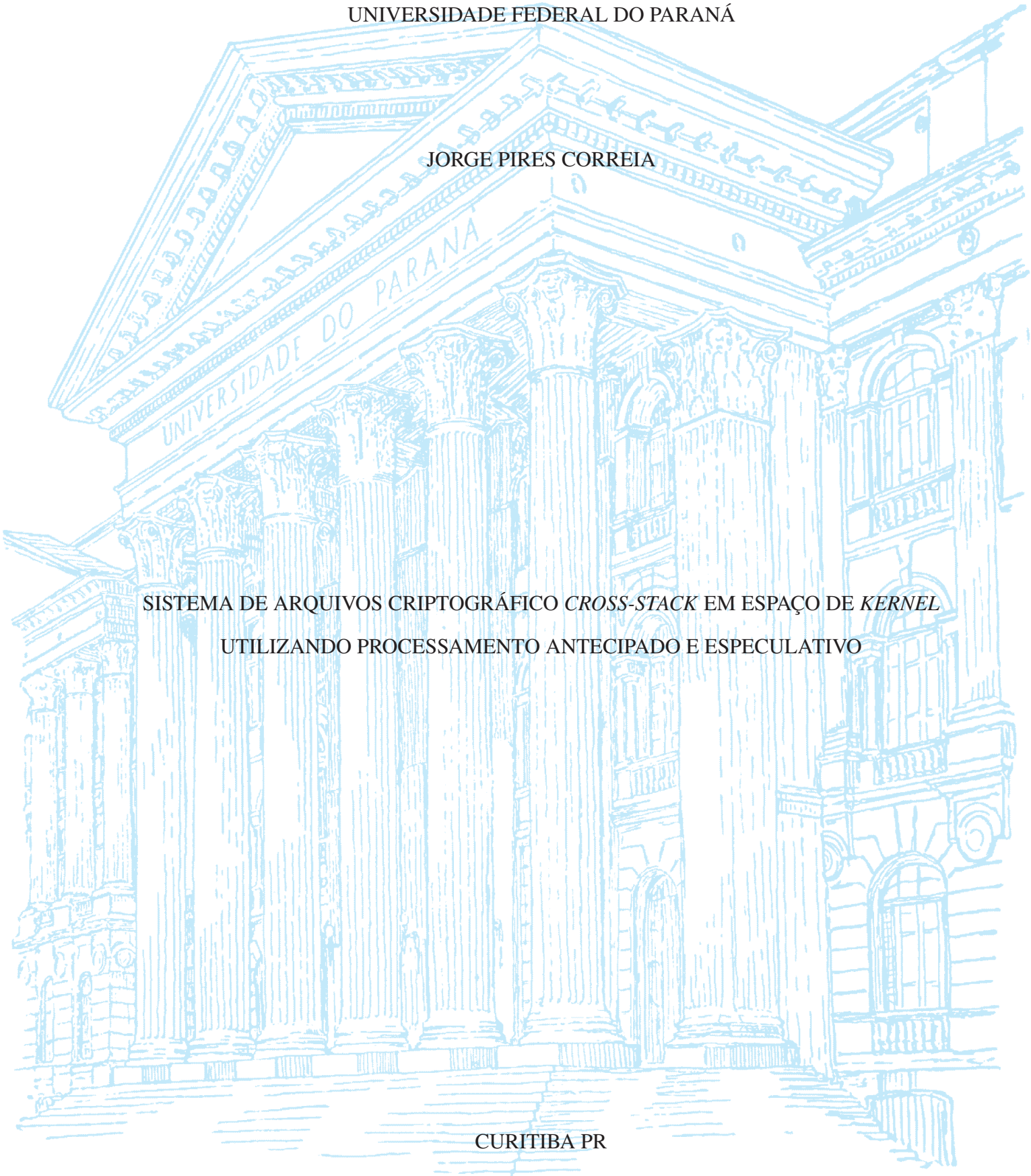
UNIVERSIDADE FEDERAL DO PARANÁ

JORGE PIRES CORREIA




# SISTEMA DE ARQUIVOS CRIPTOGRÁFICO *CROSS-STACK* EM ESPAÇO DE *KERNEL* UTILIZANDO PROCESSAMENTO ANTECIPADO E ESPECULATIVO

CURITIBA PR

2023

JORGE PIRES CORREIA

# SISTEMA DE ARQUIVOS CRIPTOGRÁFICO *CROSS-STACK* EM ESPAÇO DE *KERNEL* UTILIZANDO PROCESSAMENTO ANTECIPADO E ESPECULATIVO

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: Wagner M. Nunan Zola.

CURITIBA PR

2023

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

Correia, Jorge Pires
Sistema de arquivos criptográfico Cross-Stack em espaço de Kernel utilizando processamento antecipado e especulativo / Jorge Pires Correia. – Curitiba, 2023.
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. Criptografia de dados (Computação). 2. Arquivos de computador. 3. Espaço de kernel. 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 **JORGE PIRES CORREIA** intitulada: **Sistema de Arquivos Criptográfico Cross-Stack em Espaço de Kernel Utilizando Processamento Antecipado e Especulativo**, 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, 01 de Setembro de 2023.

Assinatura Eletrônica
06/09/2023 15:12:15.0
WAGNER MACHADO NUNAN ZOLA
Presidente da Banca Examinadora

Assinatura Eletrônica
06/09/2023 15:24:13.0
ELIAS PROCÓPIO DUARTE JÚNIOR
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
11/09/2023 11:05:08.0
JOSÉ MARCOS DA SILVA NOGUEIRA
Avaliador Externo (UNIVERSIDADE FEDERAL DE MINAS GERAIS)


Documento assinado eletronicamente de acordo com o disposto na legislação federal Decreto 8539 de 08 de outubro de 2015.
Gerado e autenticado pelo SIGA-UFPR, com a seguinte identificação única: 312943
**Para autenticar este documento/assinatura, acesse https://siga.ufpr.br/siga/visitante/autenticacaoassinaturas.jsp e insira o codigo 312943**

*Aos meus pais, Jornandes e Edna.*

## AGRADECIMENTOS

Gostaria de agradecer a Deus, inteligência suprema e causa primária de todas as coisas, pelas incontáveis oportunidades e pelo puro amor. Agradeço também ao Mestre Jesus e aos espíritos amigos pelo auxílio constante e inigualável.

À minha mãe, Edna, pela dedicação, exemplo e êxito no papel materno.

Ao meu pai, Jornandes, pelo suporte inquestionável, lições valiosas e esforço imensurável para que eu pudesse desfrutar das melhores condições possíveis.

À minha irmã, Lorena, pela companhia diária e pela ajuda ímpar.

À minha namorada Beatriz, que mesmo do outro lado do país se fez presente em todos os momentos dessa trajetória, me ajudando e incentivando de forma indescritível.

Ao professor Wagner, pelo empenho, dedicação e contribuições que permitiram a realização deste trabalho.

A João, pelo companheirismo de longa data e pelos aprendizados que me permitiram chegar até aqui.

Aos colegas de laboratório Fabrício e Vinícius, que além da companhia, viabilizaram diversas oportunidades que foram, e ainda são, de grande importância para mim.

Por fim, gostaria de agradecer às demais pessoas que contribuíram das mais diversas formas, desde auxílios diretos até pensamentos positivos. Eu não teria chegado até aqui sem vocês.

Obrigado do fundo do coração.

## RESUMO

Sistemas de Arquivos Criptográficos (SACs) proporcionam confidencialidade e privacidade aos dados armazenados submetendo-os a operações criptográficas. Juntamente com essa segurança, existe a perda de desempenho inerente ao processamento dos dados. Durante as operações de E/S, a CPU fica ociosa enquanto o dispositivo de armazenamento está movendo os dados requisitados. Esse tempo ocioso pode ser aproveitado para realizar operações criptográficas antecipadas e especulativas, reduzindo consideravelmente a latência das operações do SAC. Trabalhos anteriores demonstraram que essa abordagem gera ganhos de desempenhos consideráveis, contudo, eles não exploraram toda a capacidade dessa ideia, tendo em vista que os resultados foram gerados através de abordagens empilhadas e em espaço de usuário. Dessa forma, as arquiteturas previamente propostas não provem integração com os diferentes níveis da pilha de armazenamento, impossibilitando potenciais otimizações. Neste trabalho, as diversas possibilidades arquiteturais providas por uma abordagem *cross-stack* foram exploradas para propor um novo SAC em espaço de *kernel* completamente integrado com o subsistema de armazenamento. Dessa forma, é possível executar operações antecipadas de forma otimizada. Os resultados, gerados a partir de uma prova de conceito implementada através de *hooks* no sistema de arquivo EXT4, indicam uma queda significativa na latência das operações do SAC proposto, assim como um aumento na vazão de dados quando comparado com o sistema de arquivos criptográfico padrão do Linux, especialmente para dispositivos de armazenamento de baixa latência.



Palavras-chave: Sistemas de arquivos criptográficos. Criptografia antecipada e especulativa. Espaço de kernel.

## ABSTRACT

Cryptographic File Systems (CFSs) provide confidentiality and privacy of stored data by subjecting it to cryptographic operations. Alongside this security, there is a performance penalty caused by the data processing. During I/O operations, the CPU becomes idle while the storage device is moving the requested data. This CPU idle time can be used to perform ahead-of-time and speculative encryption, considerably reducing the latency of CFS operations. Previous works have shown that this approach generates significant performance improvements, however, they did not explore the full capacity of this idea, considering that the results were generated through stacked and user-space approaches. Therefore, the previously proposed architectures do not provide integration with different levels of the storage stack, missing potential optimizations. In this work, the diverse architectural possibilities provided by a cross-stack approach were explored to propose a new in-kernel CFS completely integrated with the storage subsystem. Then, it is able to perform optimized ahead-of-time operations. The results, generated by a proof-of-concept implementation through hooks in the EXT4 file system, showed a significant decrease in the latency of the proposed CFS operations, as well as an increase in data throughput when compared to the default Linux CFS, especially for low-latency storage devices.



Keywords: Cryptographic file system. Ahead-of-time and speculative encryption. Kernel space.

## LISTA DE FIGURAS


2.1 Visão geral da (a) encriptação e (b) decriptação do algoritmo AES com chave de 128 bits (Stallings, 2017). . . . . 17
2.2 Encriptação nos modos de operação CBC, XTS e CTR . . . . . 19
2.3 Visão geral dos componentes de armazenamento do Linux. . . . . 21
2.4 Exemplo do layout de um *block group*. . . . . 24
2.5 Exemplo de mapeamento indireto de blocos de dados. Adaptado de Carrier (2005). 25
2.6 Exemplo de mapeamento através de *extents* (Mathur et al., 2007).. . . . . 25
2.7 Visão geral da estrutura `bio` (Love, 2010).. . . . . 26

3.1 Arquitetura do CryptFS (Zadok et al., 1998).. . . . . 29
3.2 Formato de arquivo do eCryptFS (Halcrow, 2005). . . . . 30
3.3 Arquitetura do NV-eCryptFS (Xiao et al., 2018). . . . . 31
3.4 Arquitetura do SafeFS (Pontes et al., 2017). . . . . 34

4.1 Fluxo simplificado de uma operação de leitura ou escrita do EncFS++. . . . . 36
4.2 (a) Formato do arquivo global. (b) *Nonce node* (*N-node*). (c) Formato do arquivo dedicado de *nonces*. (Eduardo et al., 2019). . . . . 36
4.3 Gerenciamento de contextos criptográficos no Encfs++ (Eduardo et al., 2019). . . 37
4.4 Execução da operação de leitura de um SAC convencional. . . . . 38
4.5 Gerenciamento do *Global File*. . . . . 41
4.6 Estrutura do Global Counter (Eduardo et al., 2019). . . . . 43
4.7 Arquitetura das operações de escrita. . . . . 45
4.8 Arquitetura das operações de leitura. . . . . 47

5.1 Leitura sequencial (dispositivo NVMe). . . . . 54
5.2 Leitura sequencial (dispositivo de blocos RAM). . . . . 55
5.3 Leitura randômica. . . . . 57
5.4 Escrita sequencial. . . . . 58
5.5 Escrita randômica. . . . . 59
5.6 Cargas de leitura para múltiplos clientes.. . . . . 61
5.7 Cargas de trabalho de *macro-benchmark*.. . . . . 62

## LISTA DE TABELAS


2.1 Limites do sistema de arquivos EXT4 (Kernel Development Community, 2023c). 23

4.1 Tempo de E/S para ler 4096 bytes de um sistema de arquivos armazenado em RAM e em NVMe. A requisição foi realizada utilizando a flag O_DIRECT. A RAM utilizada possui 16GB, 2933MHz, DDR4. O NVMe utilizado foi um ADATA SX6000LNP.. . . . . 39

4.2 Tempo gasto para realizar o XOR entre dois blocos de 4096 bytes e para criar uma máscara criptográfica de 4096 bytes. O XOR foi realizado utilizando instruções AVX e as máscaras foram criadas utilizando o a Linux Kernel Crypto API. O processador utilizado foi um Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz, com suporte AES-NI habilitado.. . . . . 39

5.1 Tempo de E/S para requisições de leitura sequenciais no EXT4james (dispositivo NVMe). . . . . 54

5.2 Tempo de E/S para requisições de leitura sequenciais no EXT4james (dispositivo de blocos RAM). . . . . 56

5.3 Tempo de E/S para requisições de leitura randômica no EXT4james (dispositivo NVMe). . . . . 57

5.4 Porcentagem (%) de melhora da vazão do EXT4james em relação ao fscrypt com AES-NI desabilitado em dispositivos NVMe. . . . . 60

5.5 Porcentagem (%) de melhora da vazão do EXT4james em relação ao fscrypt com AES-NI desabilitado em dispositivos RAM . . . . . 60

5.6 Porcentagem (%) de melhora da latência do EXT4james em relação ao fscrypt com AES-NI desabilitado em dispositivos NVMe . . . . . 60

5.7 Porcentagem (%) de melhora da latência do EXT4james em relação ao fscrypt com AES-NI desabilitado em dispositivos RAM. . . . . 60

## LISTA DE ACRÔNIMOS

| | |
|---|---|
| AES | Advanced Encryption Standard |
| AES-NI | Advanced Encryption Standard New Instructions |
| API | Application Program Interface |
| AVX | Advanced Vector Extensions |
| BIO | Block I/O |
| CBC | Cipher Block Chaining |
| CFS | Cryptographic File System |
| CPU | Central Processing Unit |
| CTR | Counter |
| E/S | Entrada/Saída |
| FIO | Flexible I/O |
| FUSE | File System in User Space |
| GDT | Global Descriptor Table |
| GiB | Giga Byte |
| GPU | Graphics Processing Unit |
| HDD | Hard Disk Drive |
| HMAC | keyed-Hash Message Authentication Code |
| I/O | Input/Output |
| IV | Initialization Vector |
| KiB | Kilo Byte |
| LKM | Loadable Kernel Module |
| LUKS | Linux Unified Key Setup |
| MiB | Mega Byte |
| NAS | Network Attached Storage |
| NIST | National Institute of Standards and Technology |
| NVMe | Non-Volatile Memory Express |
| OpenCL | Open Computing Language |
| PAM | Pluggable Authentication Module |
| PCIe | Peripheral Component Interconnect Express |
| RAM | Random Access Memory |
| SAC | Sistema de Arquivos Criptográfico |
| SSD | Solid State Disk |
| SIMD | Single Instruction Multiple Data |
| TPC | Transaction Processing Performance Council |
| TPM | Trusted Platform Module |

| | |
|---|---|
| VFS | Virtual File System |
| WAES | Warped AES |
| XTS | XEX-based Tweakable codebook mode with ciphertext Stealing |
| YCSB | Yahoo! Cloud Serving Benchmark |

**SUMÁRIO**


**1 INTRODUÇÃO . . . . . 13**
1.1 PROPOSTA . . . . . 14
1.2 CONTRIBUIÇÕES . . . . . 14
1.3 ORGANIZAÇÃO DO TRABALHO . . . . . 15
**2 FUNDAMENTAÇÃO TEÓRICA . . . . . 16**
2.1 ADVANCED ENCRYPTION STANDARD E PRINCIPAIS MODOS DE OPERAÇÕES . . . . . 16
2.2 WARPED AES E WAESLIB . . . . . 18
2.3 *ADVANCED VECTOR EXTENSIONS* E AES *NEW INSTRUCTIONS* . . . . . 20
2.4 LINUX CRYPTO API . . . . . 20
2.5 SUBSISTEMAS DE ARMAZENAMENTO DO *KERNEL* LINUX . . . . . 20
2.5.1 Virtual File System . . . . . 21
2.5.2 Sistemas de arquivos e EXT4 . . . . . 22
2.5.3 Camada de E/S de blocos . . . . . 26
2.5.4 *Page Cache* . . . . . 27
**3 REVISÃO DA LITERATURA . . . . . 29**
3.0.1 Trabalhos implementados em espaço de *kernel* . . . . . 29
3.0.2 Trabalhos implementados em espaço de usuário . . . . . 33
3.1 CONSIDERAÇÕES FINAIS . . . . . 34
**4 PROJETO E IMPLEMENTAÇÃO. . . . . 35**
4.1 LIMITAÇÕES DO ENCFS++ E DESAFIOS DE OTIMIZAÇÃO . . . . . 35
4.2 VIABILIDADE E POTENCIAIS . . . . . 37
4.3 ARMAZENAMENTO E GERENCIAMENTO DOS METADADOS CRIPTOGRÁFICOS . . . . . 40
4.4 FLUXOS DE EXECUÇÃO . . . . . 44
4.4.1 Escrita de dados . . . . . 44
4.4.2 Leitura de dados. . . . . 46
4.4.3 Demais operações . . . . . 48
4.5 ADAPTAÇÃO DA WAESLIB E GERENCIAMENTO DOS CONTEXTOS CRIPTOGRÁFICOS . . . . . 49
**5 RESULTADOS. . . . . 53**
5.1 OPERAÇÕES DE LEITURA. . . . . 54
5.2 OPERAÇÕES DE ESCRITA . . . . . 56
5.3 AMBIENTES SEM ACELERAÇÃO DE OPERAÇÕES CRIPTOGRÁFICAS . . 58

5.4 LIMITAÇÕES DO AMBIENTE DE TESTE. . . . . 59
5.5 MÚLTIPLOS CLIENTES . . . . . 61
5.6 CARGAS DE TRABALHO DE *MACRO-BENCHMARK* . . . . . 62
**6 CONCLUSÃO . . . . . 64**
**REFERÊNCIAS . . . . . 65**

# 1 INTRODUÇÃO

Confidencialidade e privacidade são tópicos cada vez mais importantes no contexto de sistemas computacionais. Informações sensíveis armazenadas em máquinas de terceiros, como por exemplo serviços de nuvem, ou até mesmo em computadores pessoais são comumente submetidas a operações criptográficas com o objetivo de garantir os princípios em questão. Contudo, a utilização desses procedimentos são computacionalmente custosos, diminuindo o desempenho das aplicações que os utilizam, ou até mesmo tornando o seu uso inviável. Outra questão a ser considerada é que aplicações que não proveem operações criptográficas necessitam de modificações em suas arquiteturas para que possam oferecer esses serviços.

Uma forma de resolver estes problemas é implementar o suporte à criptografia dentro do sistema de arquivos, tornando-os Sistemas de Arquivos Criptográficos (SACs). Essa abordagem permite a execução transparente das operações criptográficas sem a modificação de qualquer interface exposta para o espaço de usuário, além de apresentar um potencial de otimizações, considerando o acesso a estruturas e interfaces de baixo nível.

Conceitualmente, para uma requisição de leitura, um SAC executa três operações sequenciais: (I) ler o dado do dispositivo de armazenamento; (II) decriptar o dado; (III) enviar o dado para a aplicação. Vale ressaltar que a operação (I) é uma operação de E/S (executada pelo dispositivo de armazenamento) e a operação (II) é a execução de um conjunto de instruções (realizada pela CPU). Ainda que esses dois passos sejam executados por diferentes dispositivos, eles precisam ser serializados, pois o dado precisa estar disponível antes de ser decriptado. A operação de escrita é realizada de maneira similar: (I) receber o dado da aplicação; (II) encriptar o dado; (III) escrever o dado no dispositivo de armazenamento.

A necessidade de uma execução sequencial dos passos realizados pelos SACs é inerente à forma que os algoritmos criptográficos funcionam. Contudo, o Warped AES (WAES) (Zola e De Bona, 2012) propõe a utilização do *Advanced Encryption Standard* (AES) no modo de operação *Counter* (CTR) de forma paralela e especulativa acelerada por GPU. Essas duas características do WAES se encaixam perfeitamente no contexto de sistemas de arquivos: a característica especulativa permite que a carga de trabalho pesada seja executada sem a necessidade do dado estar pronto em memória, podendo inclusive ser processada enquanto a operação de E/S está em execução; a característica paralela permite que requisições concorrentes sejam executadas em diferentes núcleos de processamento ao mesmo tempo. Com o objetivo de explorar essa ideia, Eduardo et al. (2019); Eduardo (2019) mostraram a eficácia de aplicar a encriptação paralela e especulativa em sistemas de arquivos através da utilização do WAES em um SAC empilhado em espaço de usuário. Embora os resultados apresentem uma melhora de até 269% em vazão de dados e 112% na eficiência de uso da CPU quando comparado com o sistema de arquivo base Encfs (Gough, 2017), o contexto de execução e a forma de adaptação do sistema de arquivos base, apresentados a seguir, impõem restrições arquiteturais e perda de desempenho. De maneira análoga, de Andrade e Zola (2022); de Andrade (2022) modificaram a implementação do WAES para não depender de uma GPU e executar em CPUs *multicore*, também gerando resultados promissores mas mantendo o contexto de execução e a forma de adaptação do sistema de arquivos.

Existem diferentes maneiras de projetar um sistema de arquivos. As duas abordagens principais estão relacionadas ao contexto de execução: em espaço de usuário ou em espaço de *kernel*. Um sistema de arquivos em espaço de usuário normalmente utiliza uma biblioteca como FUSE (Szeredi, 2003), PUFFS (Kantee, 2007), entre outras (Schmuck e Haskin, 2002; Steere et al., 1990; Westerlund e Danielsson, 1998), que permitem a comunicação entre a implementação

em espaço de usuário e um módulo dentro do *kernel* que se comunica com as camadas inferiores. Dentre as principais vantagens da utilização desse método estão o isolamento de processos e o acesso às bibliotecas disponíveis para as aplicações em espaço de usuário, além da abstração da complexidade do *kernel*. Contudo, sistemas de arquivos em espaço de usuário sofrem com o alto número de trocas de contexto (Vangoor et al., 2017) e a falta de acesso à interfaces de baixo nível. Por outro lado, sistemas de arquivos em espaço de *kernel* podem acessar todos os subsistemas do *kernel*, o que habilita diferentes tipos de otimização.

Um SAC é normalmente projetado utilizando um sistema de arquivos já existente, tendo em vista que as abstrações básicas como *inodes*, diretórios e arquivos podem ser reutilizados, assim como fluxos básicos de execução. Existem duas principais abordagens relacionadas à adaptação dos sistemas de arquivos: empilhamento e *hooking*. Em uma abordagem baseada em empilhamento, o sistema de arquivos base é visto como uma caixa preta, de maneira que as novas funcionalidades constituem uma camada acima dele, o que permite abstrair seus mecanismos. Por outro lado, sistemas de arquivos baseados em *hooks* são implementados através de chamadas de funções (*hooks*) inseridas dentro do código fonte dos sistemas de arquivos, proporcionando controle total sobre os fluxos de execuções, estruturas e interfaces.

## 1.1 PROPOSTA

Levando em conta o potencial da utilização de criptografia especulativa e paralela no contexto de sistemas de arquivos, é perceptível que a sua aplicação em um contexto favorável pode possibilitar ganhos de desempenho consideráveis, além de permitir arquiteturas completamente adaptadas ao contexto de execução, objetivando otimizações.

Este trabalho busca tirar proveito do acesso à pilha de armazenamento provido por uma execução em *kernel* para projetar um SAC *cross-stack*, ou seja, um SAC que esteja ciente das operações realizadas em cada nível da pilha de armazenamento. Para isso, é necessário utilizar uma abordagem baseada em *hooks* para adaptar o sistema de arquivos base, além de funcionalidades de hardware como AVX (Lomont, 2011) e AES-NI (Gueron, 2010) para aplicar criptografia especulativa em sistemas de arquivos. Através dessa abordagem, um novo conjunto de possibilidades relacionadas à arquitetura do sistema se abre, permitindo a modelagem de um novo SAC completamente integrado com o subsistema de armazenamento.

## 1.2 CONTRIBUIÇÕES

Como resultado deste trabalho, são apresentadas as seguintes contribuições:

- Demonstração de como cada nível da pilha de armazenamento do *kernel* Linux pode contribuir para o gerenciamento de metadados e pontos de execução de funções criptográficas;
- Apresentação de uma arquitetura que integre os componentes presentes em diferentes níveis da pilha de armazenamento;
- Implementação de um SAC prova-de-conceito a partir da modificação do sistema de arquivos padrão do Linux, ext4 (Kernel Development Community, 2023d), com base na arquitetura proposta;
- Validação da proposta através da comparação do SAC implementado com o SAC disponibilizado pelo *kernel* Linux, analisando quesitos como vazão, tempo de execução e latência em diferentes cenários.

A implementação da prova-de-conceito será disponibilizada em um diretório público. Além das contribuições citadas anteriormente, este trabalho inspirou o trabalho publicado e apresentado no Computer On The Beach 2022 (Correia e Zola, 2022). A proposta deste trabalho foi publicada e apresentada na XXIII Escola Regional de Alto Desempenho da Região Sul (ERAD/RS 2023) (Correia e Zola, 2023). Os resultados finais estarão em artigo que está em preparação para submissão no USENIX FAST 2024.

## 1.3 ORGANIZAÇÃO DO TRABALHO

O restante deste trabalho está organizado da seguinte maneira: o Capítulo 2 apresenta os fundamentos teóricos que baseiam este trabalho; o Capítulo 3 apresenta uma revisão de literatura relacionada a sistemas de arquivos criptográficos, suas arquiteturas e contextos de execução; o Capítulo 4 descreve este trabalho, enfatizando o projeto e implementação da arquitetura proposta; o Capítulo 5 apresenta os resultados alcançados com a proposta, assim como a comparação com o estado da arte; o Capítulo 6 apresenta as considerações finais, assim como trabalhos futuros que permitirão estender e aperfeiçoar as ideias aqui expostas.

# 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os fundamentos teóricos que sustentam este trabalho. Inicialmente, o algoritmo de cifragem de blocos AES é brevemente apresentado, assim como os principais modos de operação que podem ser utilizados. O Warped AES, juntamente com a sua implementação WAESlib são introduzidos. As tecnologias de hardware AVX e AES-NI são expostas, bem como a biblioteca criptográfica Crypto API presente no *kernel* Linux. Por fim, o subsistema de armazenamento do *kernel* Linux é discutido, enfatizando suas camadas e componentes.

## 2.1 ADVANCED ENCRYPTION STANDARD E PRINCIPAIS MODOS DE OPERAÇÕES

Advanced Encryption Standard (AES) (Dworkin et al., 2001) é a especificação padrão de encriptação de dados definida pelo Instituto Nacional de Padrões e Tecnologia dos Estados Unidos (NIST). O AES é um algoritmo de encriptação de chave simétrica, capaz de encriptar/decriptar dados de 128 bits, podendo utilizar três tamanhos distintos de chaves: 128 bits, 192 bits e 256 bits. O tamanho da chave é responsável por indicar a quantidade de rodadas que são realizadas durante o processo de encriptação/decriptação: uma chave de 128 bits implica na execução de 10 rodadas; uma chave de 192 bits implica na execução de 12 rodadas; uma chave de 256 bits implica na execução de 14 rodadas.

A Figura 2.1 apresenta o funcionamento do AES. Inicialmente, os 128 bits a serem encriptados ou decriptados são dispostos em um vetor bidimensional agrupados em bytes, formando uma matriz 4x4 denominada `estado`. Cada rodada, com exceção da última, é composta por quatro etapas: (I) substituição de bytes, em que cada byte é substituído por um outro de acordo com a tabela denominada `lookup table`; (II) deslocamento de linhas, em que as três últimas colunas são ciclicamente deslocadas; (III) mistura de colunas, em que os quatro bytes de cada coluna são combinados entre si; (IV) adição da chave da rodada, em que uma chave de 128 bits específica para essa rodada, derivada da chave fornecida para o algoritmo, é combinada com a matriz `estado`. A última rodada não realiza a etapa de mistura de colunas.

Cada um dos passos executados pelo AES tem como objetivo tornar complexa a relação entre a chave utilizada e o texto cifrado, além de esconder a ocorrência de determinados símbolos recorrentes no texto claro no texto cifrado, dificultando análises estatísticas. Essas propriedades, aplicadas sucessivamente em rodadas, constituem o funcionamento das principais cifras simétricas atuais (Shannon, 1949; Stallings, 2017).

Tendo em vista que o AES só é capaz de encriptar/decriptar blocos de dados de tamanho fixo de 128 bits, diversos modos de operação foram desenvolvidos para permitir a aplicação do algoritmo em questão em dados de tamanhos arbitrários mantendo as propriedades de segurança (Dworkin, 2001; Ferguson et al., 2011). Cada modo de operação possui características individuais e formas diferente de se aplicar o AES. A encriptação de dados armazenados (*data at rest*) é um cenário comum e, por isso, existem alguns princípios que direcionam a aplicação dos modos de operação neste contexto (Menezes et al., 2018), como por exemplo:

- Os dados no dispositivo de armazenamento devem permanecer confidenciais;
- Leitura e escrita de dados ao disco devem ser rápidas, independente do local de armazenamento dentro do disco;

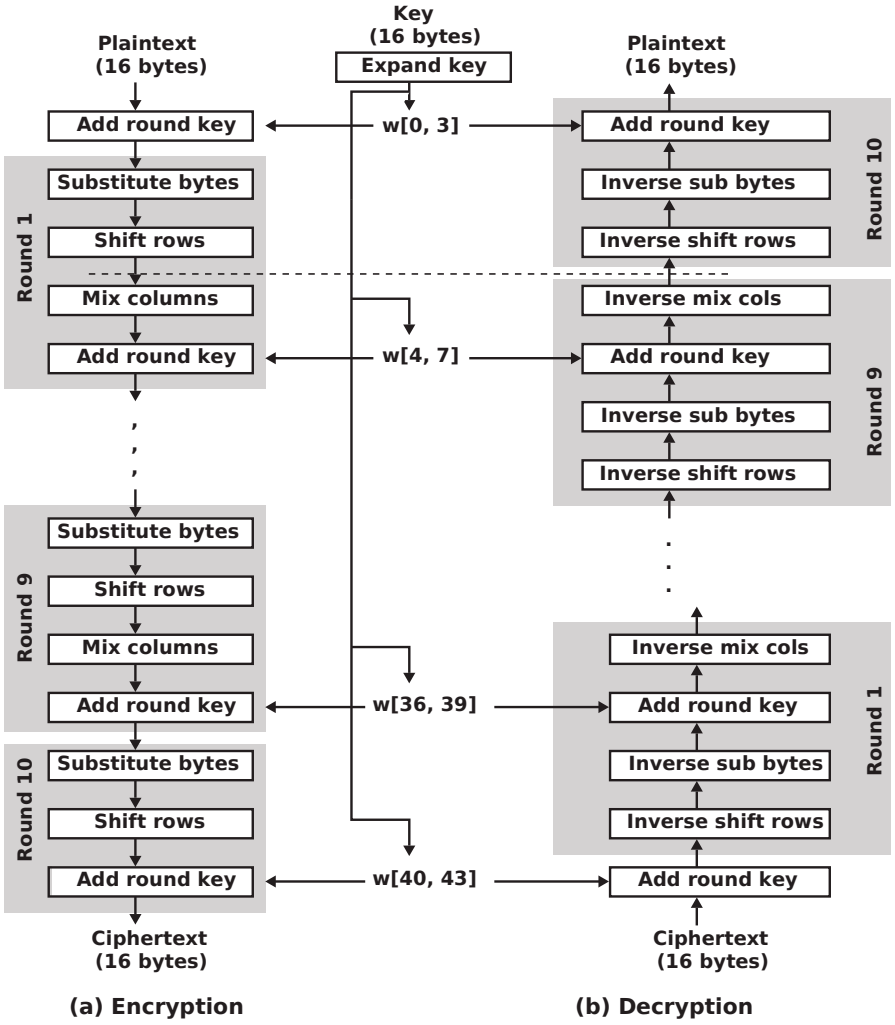



Figura 2.1: Visão geral da (a) encriptação e (b) decriptação do algoritmo AES com chave de 128 bits (Stallings, 2017).

- Os metadados criptográficos não devem ocupar um tamanho considerável quando comparado com os dados;
- O atacante pode ter acesso total aos dados brutos do dispositivo de armazenamento.

O modo de operação *Cipher Block Chaining* (CBC), mostrado na Figura 2.2a, encadeia os blocos de dados, de forma que, para encriptar o bloco N, é necessário o bloco encriptado N-1. Um Vetor de Inicialização (IV) imprevisível é utilizado para encriptar o primeiro bloco, já que não existe um bloco anterior. Esse modo de operação não permite encriptação paralela de blocos visto que existe uma dependência entre eles.

O modo de operação *XEX-based tweaked-codebook mode with ciphertext stealing* (XTS), apresentado na Figura 2.2b, é comumente utilizado em sistemas de encriptação de todo o disco, visto que ele permite lidar com setores de tamanho não divisível pelo tamanho do bloco (National Institute of Standards and Technolog, 2023). Cada setor é uma agregação de blocos de 128 bits, de forma que o número do setor e o índice do bloco de 128 bits são utilizados como parâmetro para as operações. Cada bloco é processado independentemente, possibilitando a encriptação e decriptação de forma paralela.

O modo de operação *Counter* (CTR), mostrado na Figura 2.2c, processa cada bloco individualmente, aplicando a cifra de bloco em um contador de 128 bits exclusivo do bloco

em questão. O resultado é submetido a uma operação XOR com o bloco de dados, gerando o bloco cifrado. O contador pode ser concatenado com um IV aleatório, gerando um *nonce* que é utilizado somente uma vez para uma determinada chave de encriptação. Assim como o XTS, o modo de operação CTR é capaz de encriptar e decriptar blocos diferentes ao mesmo tempo.

Todos os modos de operação aqui citados são comprovadamente seguros, desde que aplicados corretamente (Lipmaa et al., 2000; Martin, 2010; Dworkin, 2001; Van Tilborg e Jajodia, 2014; Paar e Pelzl, 2009).

## 2.2 WARPED AES E WAESLIB

Analisando a Figura 2.2c, é perceptível que a carga de trabalho pesada das operações criptográficas no modo de operação CTR são realizadas na parte pontilhada, sendo seus parâmetros o *nonce*/contador e a chave. Esse primeiro processamento gera uma máscara criptográfica, que será aplicada ao bloco de dados através de uma operação XOR, gerando o dado cifrado. O WAES (Zola e De Bona, 2012) tira proveito desse modo de operação através da utilização da GPU para prover alta vazão de máscaras especulativamente, possibilitando a aplicação destas pela CPU quando os dados estiverem prontos em memória. Juntamente com essa vantagem, O WAES provê melhor utilização da CPU, visto que esta fica livre executar outras tarefas enquanto as máscaras estão sendo criadas. Como a aplicação das máscaras é rápida, ela pode ser realizada pela CPU e evitar a comunicação GPU-CPU.

A latência da transferência de dados entre a CPU e a GPU influencia consideravelmente a performance geral, visto que essa comunicação ocorre através do barramento PCI Express (PCIe). Para minimizar o número de comunicações entre esses dois componentes, o WAES propõe a agregação de requisições sequenciais, otimizando a utilização da largura de banda do PCIe.

Essa proposta gerou a implementação da WAESlib, uma biblioteca implementada em espaço de usuário que dispara *kernels* CUDA (Nickolls et al., 2008) para produção das máscaras assincronamente. Com o objetivo de facilitar o gerenciamento da requisição de criação de máscaras e, posteriormente, a sua aplicação, a WAESlib provê o conceito de contexto criptográfico.

Para cada contexto, existe uma máscara, um *nonce* e uma chave. A máscara presente no contexto da WAESlib é, na realidade, uma abstração que pode ser a concatenação de várias máscaras de 128 bits do AES. Na inicialização da biblioteca, é possível definir o tamanho destas máscaras, além da quantidade de contextos que a biblioteca gerenciará. A WAESlib disponibiliza uma interface composta de seis funções: `WAES_init()`, `WAES_setkey()`, `WAES_ctx()`, `WAES_encrypt()`, `WAES_decrypt()` e `WAES_finish()`.

As funções `WAES_init()` e `WAES_finish()` são responsáveis por inicializar e finalizar, respectivamente, as *threads* trabalhadoras e alocações de memória da biblioteca. A `função WAES_setkey` é utilizada para incluir uma chave de encriptação na biblioteca, que pode ser referenciada posteriormente a partir de um ID retornado por essa função. A função `WAES_ctx()` é responsável por disparar um *kernel* CUDA de forma assíncrona, que criará a máscara para o contexto passado como parâmetro. As funções `WAES_encrypt()` e `WAES_decrypt()` são responsáveis por executar o XOR entre a máscara presente no contexto definido no parâmetro que foi previamente criada e um dado, também passado por parâmetro.

É difícil prever a ordem de criação das máscaras, visto que elas são criadas de forma assíncrona pela GPU. Porém, os dados são requisitados em uma ordem bem definida e, por isso, as máscaras precisam estar prontas na mesma ordem. Para solucionar este problema, a WAESlib permite que uma prioridade seja definida ao contexto na função `WAES_ctx()`, de forma que

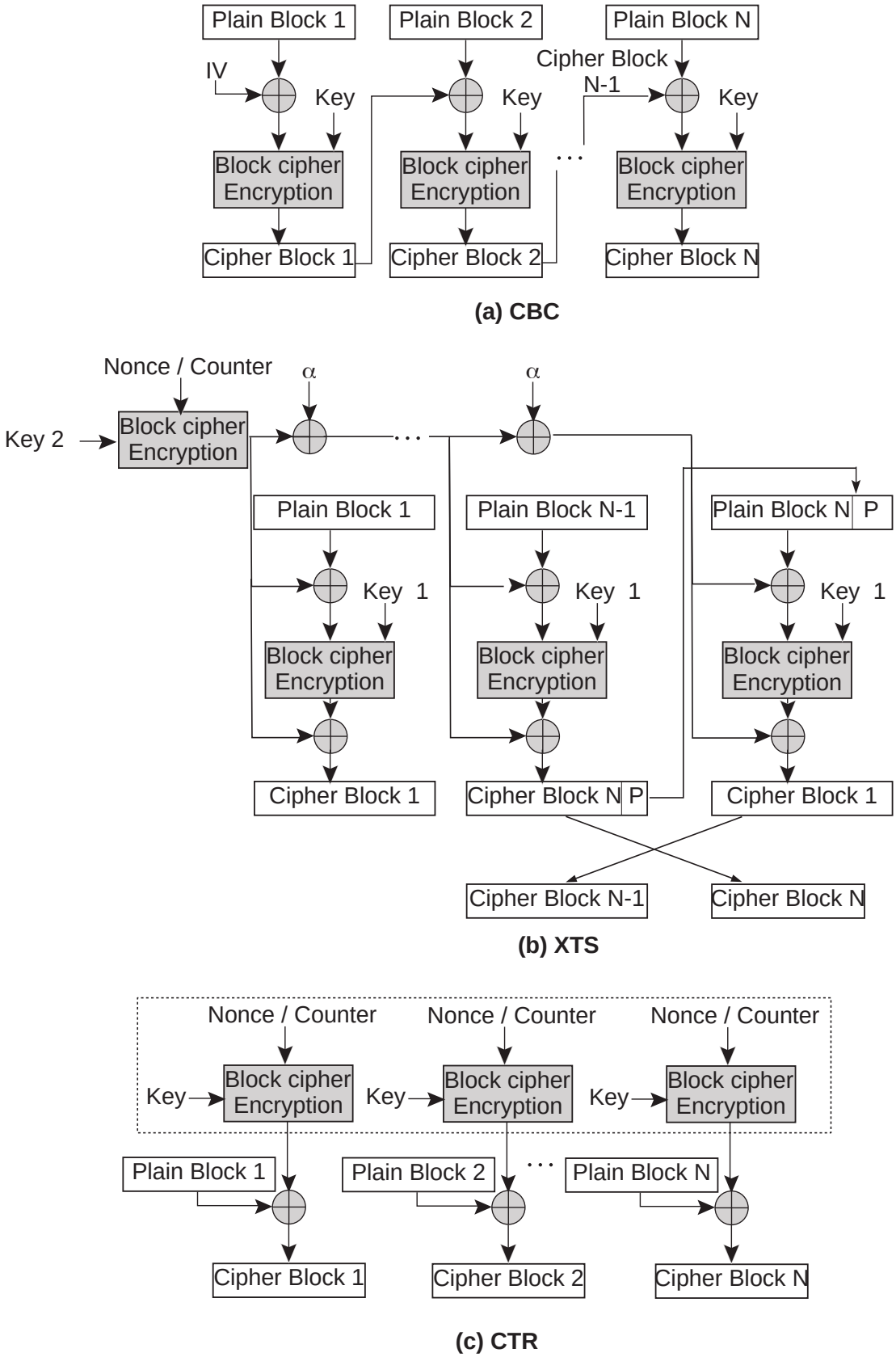



Figura 2.2: Encriptação nos modos de operação CBC, XTS e CTR

o cliente da WAESlib (o sistema de arquivos, nesse caso) produza máscaras em uma ordem

compatível com a ordem de requisição de dados, fazendo com que esta esteja pronta quando necessário.

## 2.3 *ADVANCED VECTOR EXTENSIONS* E AES *NEW INSTRUCTIONS*

Intel *Advanced Vector Extensions* (AVX) (Lomont, 2011) é uma aplicação do conceito *Single Instruction Multiple Data* (SIMD), que busca aumentar a vazão de dados processados pela CPUs. Esse objetivo é alcançado através de novas instruções e registradores que lidam com dados maiores. AVX-512 (Reinders, 2013; Kusswurm, 2022), por exemplo, oferece registradores vetoriais de 512 bits, instruções de conversão de dados, e operações *scatter*, assim como compatibilidade com as versões anteriores do AVX sem perda de performance (Konsor, 2011).

A Intel também oferece suporte em hardware para executar operações AES com o conjunto de instruções denominado AES *New Instructions* (AES-NI) (Gueron, 2010; Rott, 2012). Este conjunto é composto por seis instruções que lidam com as rodadas de encriptação e decriptação do AES, e operações com chaves. Comparado com implementações em software, o AES-NI pode melhorar a performance em até 10x para modos de operações paralelos como o CTR, além de diminuir o risco de ataques *side-channel*, visto que as operações são executadas em hardware.

## 2.4 LINUX CRYPTO API

O *kernel* Linux disponibiliza uma API para operações de criptografia denominada Crypto API (Mueller e Vasut, 2023). Além de prover uma interface para diferentes tipos e implementações de algoritmos de criptografia, a Crypto API também disponibiliza interfaces para utilização de algoritmos de compressão de dados, CRC, geração de números randômicos, entre outros. Por isso, a API intitula genericamente as operações como `transformações`. A utilização das funcionalidades da Crypto API para cifras de bloco simétricas pode ser dividida nas seguintes etapas: (I) alocação e inicialização de uma transformação com a operação desejada; (II) inserção da chave utilizada para a transformação; (III) alocação e inicialização de uma estrutura requisição, que contém informações como a transformação, buffers, *nonces*/IVs e tamanho do dado; (IV) submissão de uma requisição; (V) liberação das estruturas alocadas.

Algoritmos de encriptação de bloco podem ser utilizados diretamente, ou através de *templates*, os quais permitem a utilização dos modos de operação como CTR. Diversas implementações para uma cifra são disponibilizadas pela API. Por exemplo, dependendo da arquitetura de hardware em que o *kernel* está sendo executado, o algoritmo AES pode ser disponibilizado através de uma implementação que utiliza AES-NI, através de uma implementação em assembly, ou através de uma implementação em C. As implementações são classificadas de acordo com seu desempenho, de forma que a implementação mais rápida tem maior prioridade de execução. Caso nenhuma implementação seja especificada, a implementação de maior prioridade será utilizada. A Crypto API também fornece uma interface para que possam ser registradas novas implementações de novos algoritmos ou algoritmos existentes.

## 2.5 SUBSISTEMAS DE ARMAZENAMENTO DO *KERNEL* LINUX

O subsistema de armazenamento do Linux é composto por diferentes níveis de componentes. As chamadas de sistemas relacionadas à sistemas de arquivos são inicialmente recebidas pelo Virtual File System (VFS). Logo depois, são repassadas para o sistema de arquivos em questão, e caso necessário, requisições de E/S são submetidas à camada de E/S de blocos, que

é encarregada de realizar as requisições aos *drivers* (Tanenbaum, 2014; Bovet e Cesati, 2005; Love, 2010) . A visão geral pode ser vista na Figura 2.3.

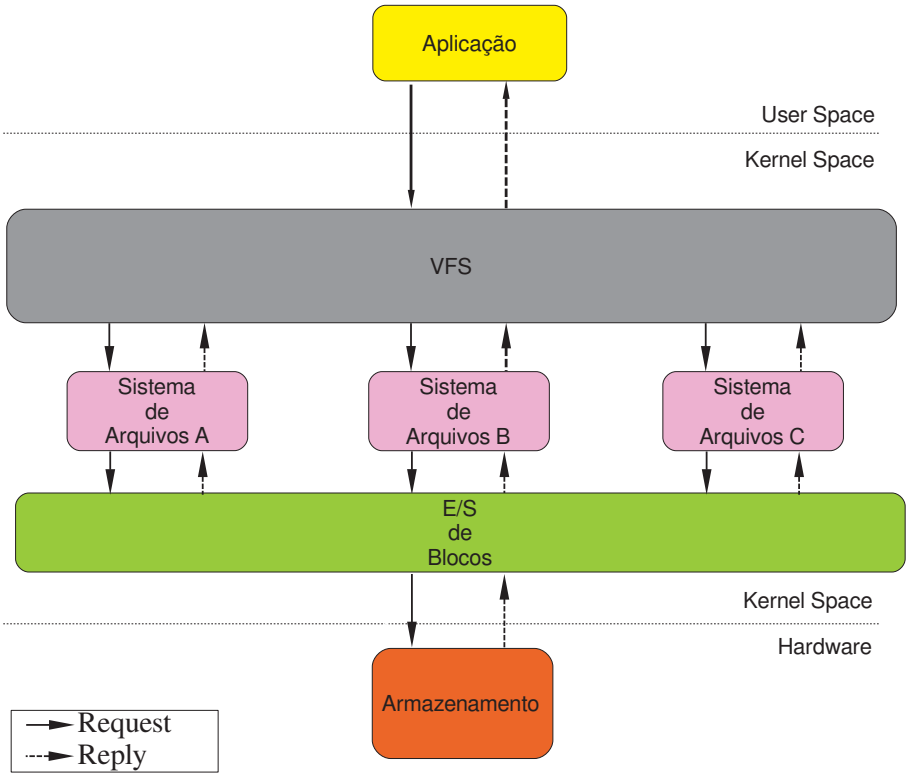



Figura 2.3: Visão geral dos componentes de armazenamento do Linux.

### 2.5.1 Virtual File System

O Virtual File System é uma camada de software que serve como interface tanto para o espaço de usuário quanto para os sistemas de arquivos. Para o espaço de usuário, o VFS define, através de chamadas de sistema, as funções que podem ser chamadas para realizar requisições aos sistemas de arquivos. Para os sistemas de arquivos, o VFS define as funções e estruturas que eles devem prover, possibilitando a coexistência de vários sistemas de arquivos ao mesmo tempo. O VFS apresenta toda a hierarquia de arquivos para o espaço de usuário em formato de árvore, possibilitando a montagem de novos sistemas de arquivos em subárvores.

Dentre todas as estruturas que o VFS define para os sistemas de arquivos, algumas devem ser evidenciadas, tais como *super block*, *inode*, *file* e *dentry*, descritas a seguir.

O *super block* é responsável por definir um sistema de arquivos que está montado. Essa estrutura define informações importantes como tipo do distema de arquivos, tamanho de blocos, localização do *inode root*, entre outras informações. O VFS requer também que os sistemas de arquivos forneçam funções que manipulam os seus *super blocks* através de uma estrutura denominada `super_operations`. Através dela, o VFS sabe quais funções chamar em operações como alocação e destruição de *inodes*, persistir um *inode*, liberar um *super block* (em operações de *umount*), requisitar informações do sistemas de arquivos, entre outras.

A estrutura *inode* é responsável por descrever arquivos, diretórios, pipes ou qualquer entidade que pode ser abstraída na forma de arquivo. O *inode* é responsável por conter informações como tipo do arquivo, tamanho do arquivo, localizações dos blocos de dados no dispositivo de armazenamento, entre outras informações. Assim como para o *super block*, o VFS requer

uma estrutura que contém as funções que são utilizadas para manipular os *inodes*, demominada `inode_operations`. Dentre as funções presentes nessa estrutura, se destacam as operações de criação e remoção de arquivos, criação de links simbólicos, criação de diretórios, entre outros.

A estrutura *file* descreve um arquivo que está aberto, de forma que existe uma estrutura *file* no *kernel* para cada descritor de arquivo em espaço de usuário. Dentre as informações presentes nessa estrutura, pode-se destacar informações como posição do cursor de leitura e escrita, modo em que o arquivo foi aberto, entre outros. A estrutura que contém as funções que manipulam a estrutura file, denominada `file_operations`, contam com funções como leitura, escrita, *seek*, abertura e fechamento de arquivos, entre outros.

A estrutura *dentry* (*directory entry*) é responsável por descrever uma entrada de diretório, principalmente relacionando um nome de arquivo (caminho na árvore) a um *inode*. Como um arquivo pode estar em mais de um lugar na árvore de arquivos criada pelo VFS (através de links), este pode assumir mais de um nome. Para que seja possível que todas as ocorrências deste arquivo compartilhem o mesmo *inode*, informações específicas de cada ocorrência são mantidas na estrutura *dentry*. As informações mais relevantes nesta estrutura são o nome da entrada de diretório, uma referencia para o *inode*, referências para a *dentry* superior (diretório em que o arquivo em questão está contido), informações como data e horário de acesso, dentre outros elementos. A estrutura `dentry_operations`, responsável por armazenar endereços de funções que manipulam as *dentry*s, contem funções para validar e *hash* de entradas de diretórios, operações de poda de um nó específico da árvore de diretórios, entre outros.

### 2.5.2 Sistemas de arquivos e EXT4

Sistemas de arquivos no *kernel* Linux formam a camada de software responsável pela política de organização dos dados do usuário nos dispositivos de armazenamento e suas requisições. Através da interface provida pelo VFS, um sistema de arquivos deve se registrar, habilitando-se a montar uma árvore de diretórios e prover suas funcionalidades para o espaço de usuário.

Antes de adentrar no funcionamento dos sistemas de arquivos em si, vale citar algumas nomenclaturas que se confundem na literatura. Dentro do sistema operacional, existem algumas unidades de tamanhos de dados que são utilizados em contextos específicos, tais como: páginas, blocos e setores. Página é uma unidade de dados utilizada pelo sistema de memória virtual do sistema operacional. Operações alinhadas ao tamanho de páginas, normalmente 4KB, conseguem atingir maiores performances que operações desalinhadas, pois o gerenciamento de endereços alinhados é otimizado em diferentes níveis. Bloco é a menor unidade endereçável do sistema de arquivos, normalmente possuindo o mesmo tamanho que uma página. Setor é a menor unidade endereçável de um dispositivo de armazenamento. Como o setor reflete uma característica física do dispositivo, normalmente seu tamanho não acompanha o tamanho de páginas do sistema operacional, sendo comum 512 bytes.

O EXT4 (Kernel Development Community, 2023d) é o sistema de arquivos padrão do *kernel* Linux. Introduzido de forma estável no *kernel* 2.6.28, a versão atual do EXT4 conta com diversos recursos (Kernel Development Community, 2023e), entre eles a capacidade de lidar com um sistema de arquivos de até 1YiB, formato *extent* de organização de blocos de dados, sistema de *journaling*, tamanhos de arquivos de até 256TiB, entre outros. A Tabela 2.1 apresenta os limites do sistema de arquivos EXT4 para diferentes tamanhos de blocos.

Os sistemas de arquivos são responsáveis por gerenciar a forma com que os dados são armazenados no dispositivo de armazenamento. O EXT4 encara o dispositivo como um *array* de blocos. Um bloco pode conter entre 1KB e 64KB, sendo sempre alinhados ao tamanho dos setores e normalmente sendo de 4KB. Blocos são também agrupados pelo EXT4 em unidades

Tabela 2.1: Limites do sistema de arquivos EXT4 (Kernel Development Community, 2023c).

| **Item** | **1KB** | **2KB** | **4KB** | **16KB** |
|---|---|---|---|---|
| 32-bit filesystem | | | | |
| Blocks | $2^{32}$ | $2^{32}$ | $2^{32}$ | $2^{32}$ |
| Inodes | $2^{32}$ | $2^{32}$ | $2^{32}$ | $2^{32}$ |
| File System Size | 4TiB | 8TiB | 16TiB | 256TiB |
| Blocks Per Block Group | 8,192 | 16,384 | 32,768 | 524,288 |
| Inodes Per Block Group | 8,192 | 16,384 | 32,768 | 524,288 |
| Block Group Size | 8MiB | 32MiB | 128MiB | 32GiB |
| Blocks Per File, Extents | $2^{32}$ | $2^{32}$ | $2^{32}$ | $2^{32}$ |
| Blocks Per File, Block Maps | 16,843,020 | 134,480,396 | 1,074,791,436 | 4,398,314,962,956 (really $2^{32}$ due to field size limitations) |
| File Size, Extents | 4TiB | 8TiB | 16TiB | 256TiB |
| File Size, Block Maps | 16GiB | 256GiB | 4TiB | 256TiB |
| 64-bit filesystem | | | | |
| Blocks | $2^{64}$ | $2^{64}$ | $2^{64}$ | $2^{64}$ |
| Inodes | $2^{32}$ | $2^{32}$ | $2^{32}$ | $2^{32}$ |
| File System Size | 16ZiB | 32ZiB | 64ZiB | 1YiB |
| Blocks Per Block Group | 8,192 | 16,384 | 32,768 | 524,288 |
| Inodes Per Block Group | 8,192 | 16,384 | 32,768 | 524,288 |
| Block Group Size | 8MiB | 32MiB | 128MiB | 32GiB |
| Blocks Per File, Extents | $2^{32}$ | $2^{32}$ | $2^{32}$ | $2^{32}$ |
| Blocks Per File, Block Maps | 16,843,020 | 134,480,396 | 1,074,791,436 | 4,398,314,962,956 (really $2^{32}$ due to field size limitations) |
| File Size, Extents | 4TiB | 8TiB | 16TiB | 256TiB |
| File Size, Block Maps | 16GiB | 256GiB | 4TiB | 256TiB |

denominadas *block groups*, normalmente contendo $2^{15}$ blocos. Como apresentado na Figura 2.4, cada *block group* pode conter as seguintes estruturas de controle:

- `Super block`: é o primeiro bloco do *block group*. Armazena informações sobre o sistema de arquivos;
- `Group Descriptor Table (GDT)`: Segue o *super block*. Armazena informações sobre onde estão alocados os *bitmaps* de blocos de dados e os *bitmaps* da tabela de *inodes* de todos os *block groups*;
- `Reserved Group Descriptor Table`: Grupo de blocos reservados para expansão do sistemas de arquivos. Quando um sistema de arquivos precisa ser aumentado, se faz necessária a utilização de mais descritores de grupos;
- `Journal Log Area`: Área reservada para a operação do sistema de *journaling*;
- `Inode Bitmap`: *Bitmap* que indica a ocupação dos *inodes* da *inode table* do *block group* em questão;
- `Data Block Bitmap`: *Bitmap* que indica a ocupação dos blocos de dados do *block group* em questão;
- `Inode table`: Blocos que contém os *inodes* do *block group* em questão;
- `Data blocks`: Blocos que contém os dados dos arquivos. Todos os blocos que não estão em nenhuma das categorias anteriores são considerados `data blocks`.

Vale ressaltar que a `Journal Log Area` só está presente no primeiro *block group*. O `super block` e a `Group Descriptor Table` são informações de controle indispensáveis para o funcionamento do EXT4. Por esse motivo, essas estruturas possuem backups em diferentes *block groups* pelo dispositivo de armazenamento, mas podem não estar presente em todos eles. Algumas opções de configuração do EXT4 podem alterar estes e outros parâmetros. O EXT4 possui também uma funcionalidade denominada *Flexible Block Groups*. Através dela, alguns *block groups* são concatenados em *block groups* lógicos, de forma que os *bitmaps* são aglutinados e mapeiam todo o *block group* lógico.

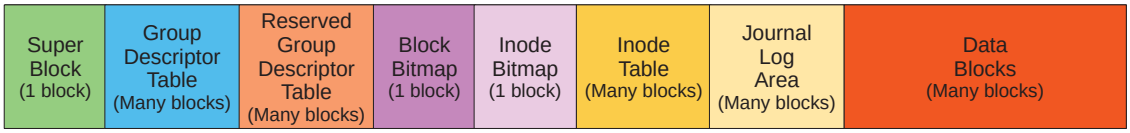



Figura 2.4: Exemplo do layout de um *block group*.

Quando alguma operação do EXT4 é requisitada, se faz necessário carregar as informações que estão no dispositivo de armazenamento para a memória. Com esse objetivo, o EXT4 define estruturas como `ext4_inode_info` e `ext4_sb_info`, que descrevem *inode* e *super block* em memória, respectivamente. Essas estruturas armazenam algumas informações que, posteriormente, serão utilizadas para preencher as estruturas fornecidas pelo VFS.

O *inode* é responsável por descrever um arquivo. Além dos metadados que o descrevem, o *inode* deve indicar onde estão localizados os seus blocos de dados no dispositivo de armazenamento. Nos sistemas de arquivos EXT2 e EXT3, antecessores do EXT4, esse gerenciamento era realizado

através do mapeamento de blocos de até três níveis, como mostra a Figura 2.5. Nesse esquema, até 12 blocos eram apontados diretamente pelo *inode* (*data block*, que apontam para blocos que contém dados). Além desses blocos, existia um ponteiro indireto de um nível, um ponteiro indireto de dois níveis e um ponteiro indireto de três níveis (*pointer blocks*, que apontam para blocos que contém referencias). Visto que esse método se mostrava ineficiente quando lidava com arquivos grandes devido a fragmentação em disco, o EXT4 introduziu o método de mapeamento denominado *Extents Tree*, ilustrado na Figura 2.6. *Extents* são organizados em formato de árvore, de forma que cada nó possui uma estrutura `ext4_extent_header`. Caso o nó em questão seja não folha, outros nós são apontados através de estruturas `ext4_extent_idx`. Caso o nó seja folha, uma estrutura `ext4_extent` aponta para blocos contíguos de dados. O mapeamento realizado desta maneira diminui a fragmentação pois lida com blocos fisicamente contíguos, além de aumentar o desempenho geral do sistema em operações em grandes arquivos.

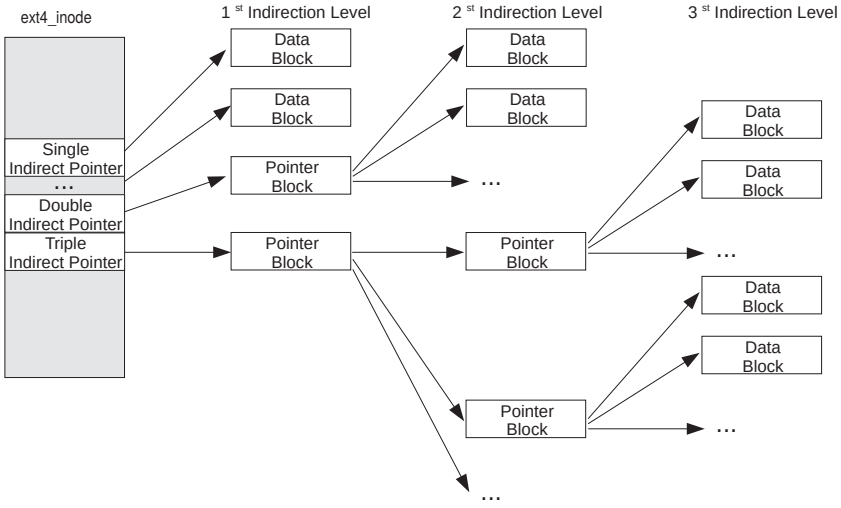



Figura 2.5: Exemplo de mapeamento indireto de blocos de dados. Adaptado de Carrier (2005).

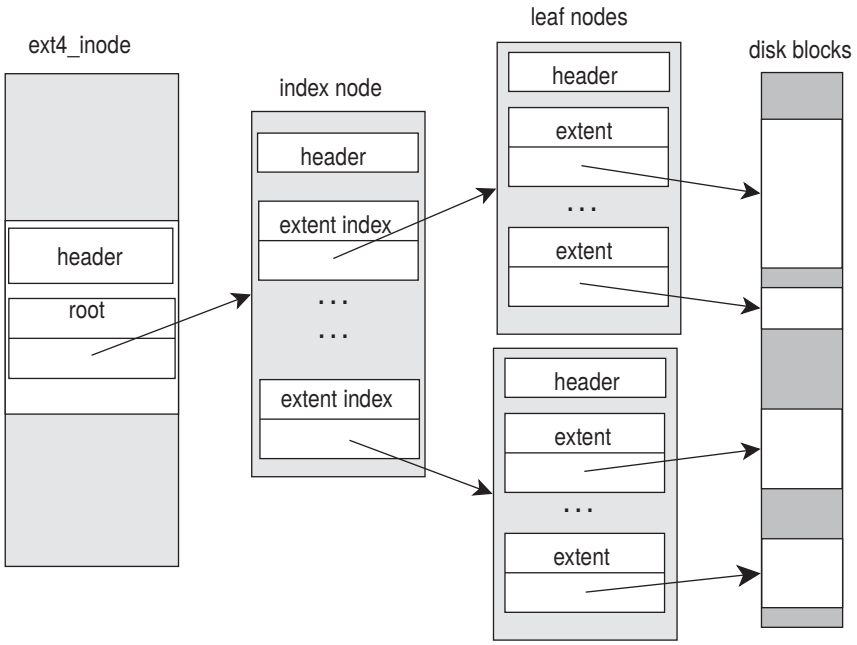



Figura 2.6: Exemplo de mapeamento através de *extents* (Mathur et al., 2007).

O EXT4 disponibiliza um meio de adicionar metadados ao *inodes*, denominado *extended attributes*. Um *extended attribute* é definido através do par nome-valor e um *namespace* que define o nível de acesso ao atributo. Caso o tamanho dos pares de atributos seja pequeno o suficiente, estes podem ser armazenados no próprio *inode*. Caso contrário, um bloco extra é alocado.

### 2.5.3 Camada de E/S de blocos

A camada de E/S de blocos fornece uma interface de acesso aos dispositivos de E/S para os sistemas de arquivos. Como cada dispositivo pode ter políticas e arquiteturas diferentes, um gerenciamento complexo e detalhado se faz necessário, afim de possibilitar que as camadas de software superiores acessem somente sua interface.

A camada de E/S de blocos do *kernel* Linux disponibiliza duas principais estruturas para as camadas superiores: `buffer_head` e `bio`. Quando um bloco do dispositivo de armazenamento é movido para a memória, um *buffer* é criado para armazená-lo. Cada *buffer* é responsável por armazenar um bloco. Tendo em vista que o sistema operacional precisa de informações para gerenciar esse *buffer*, uma estrutura `buffer_head` é anexada a cada *buffer*. Contudo, essa estrutura só consegue descrever um único bloco, forçando a divisão de grandes operações de E/S e diminuindo o desempenho. Objetivando a resolução deste problema e a simplificação do código, a estrutura `bio` foi implantada.

No *kernel* Linux atual, `bio` é a estrutura básica das operações de E/S. Essa estrutura descreve os dados envolvidos nas operações, através de uma lista de segmentos. Cada segmento é descrito por uma sequência de bytes contíguos em uma página na memória. Dessa forma, dados que são contíguos no dispositivo de armazenamento mas não contíguos em memória podem ser referenciados por uma unica estrutura `bio`. A visão geral da `bio` é representada na Figura 2.7.

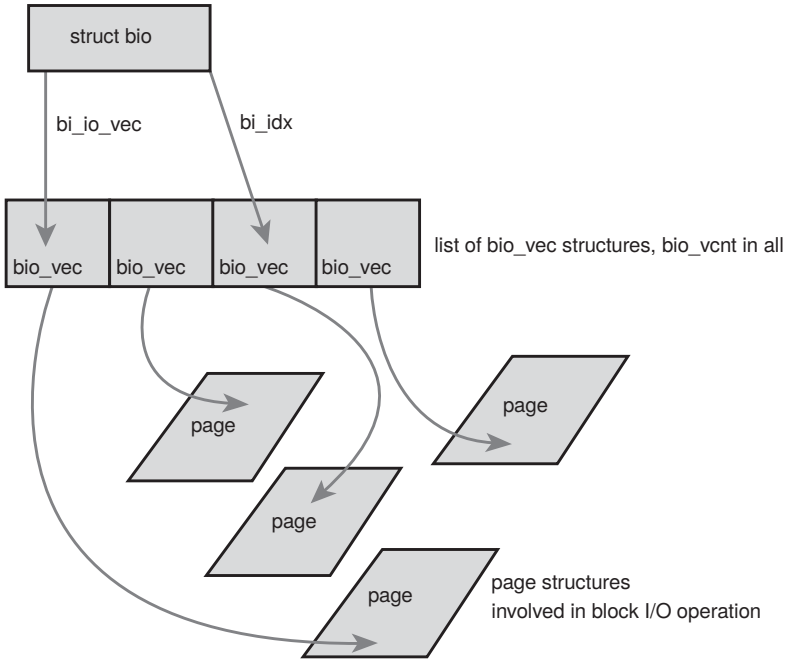



Figura 2.7: Visão geral da estrutura `bio` (Love, 2010).

A estrutura `bio` é composta, principalmente, por uma lista de estruturas `bio_vec`. Cada `bio_vec` é responsável por descrever um segmento contínuo em uma página na memória. Essa descrição se dá através de três componentes: página onde se encontra o segmento,

deslocamento dentro da página onde se inicia o segmento e tamanho do segmento. Ainda que a `bio` seja a principal estrutura que trata as operações de E/S, a estrutura `buffer_head` ainda é utilizada em determinados contextos do *kernel* Linux.

Para um sistema de arquivos executar uma operação de E/S, uma estrutura `bio` deve ser alocada e preenchida. Em seguida, a `bio` é submetido à camada de E/S de blocos. Esta camada, por sua vez, não repassa a submissão diretamente para o *driver*. Com o objetivo de diminuir a quantidade de *seeks*, que é uma das operações mais custosas principalmente para os dispositivos de armazenamento de disco, as estruturas são enfileiradas, de forma que alguns algoritmos de escalonamento de E/S realizam operações de ordenação e aglomeração das requisições.

### 2.5.4 *Page Cache*

*Page cache* é um conjunto de páginas dedicadas a preservar dados dos dispositivos de armazenamento na memória. Através da *page cache*, o sistema operacional pode diminuir consideravelmente a quantidade de operações de E/S, que são ordens de magnitude mais lentas que operações em memória.

Para uma chamada de sistema *read*, o sistema operacional verifica se o bloco requisitado está presente na *page cache*. Caso esteja presente, o dado é retornado sem a necessidade de uma operação de E/S. Caso não esteja presente, a operação é submetida à camada de E/S de blocos. Quando a operação for finalizada, o dado estará na *page cache* e será retornado ao processo que o requisitou.

Já para uma chamada de sistema *write*, existem três modos de operação de *cache*: (I) a *cache* pode ser ignorada, de forma que o dado é escrito diretamente no dispositivo de armazenamento; (II) a escrita pode ser realizada tanto na *cache* quanto no dispositivo de armazenamento, de modo que ambos contém o dado atualizado; (III) a escrita é realizada somente na *cache* e a página em questão é marcada como *dirty*, informando ao sistema operacional que o dado presente no dispositivo de armazenamento não está atualizado. Posteriormente, *threads* do *kernel* realizarão a operação de E/S. A operação (I) é indicada para processos quer não desejam cópias extras durante o processo de escrita, normalmente porque um esquema de *cache* próprio é utilizado. No Linux, a flag `O_DIRECT` definida no momento de abertura do arquivo, indica ao *kernel* a intenção de não utilizar a *page cache* e utilizar o próprio *buffer* em espaço de usuário como origem para a operação de E/S. A operação (II) é indicada para processos que desejam usufruir da *page cache* em operações de leitura, mas precisam da certeza de que quando a operação de escrita retornar, o dado estará persistido no armazenamento não volátil. Para isso, a flag `O_SYNC` deve ser indicada no momento de abertura no arquivo. A operação (III) é a padrão utilizada pelo *kernel* Linux e a que apresenta melhor vazão de dados, já que os dados são escritos na memória, resultando em uma escrita assíncrona ao armazenamento não volátil. Vale ressaltar que a utilização da flag `O_DIRECT` não implica na certeza que o dado está persistido no momento do retorno da chamada de sistema, pois pode existir sistemas de *cache* no próprio dispositivo de armazenamento. As flags `O_DIRECT` e `O_SYNC` podem ser utilizadas em conjunto, caso seja necessário.

A *page cache* não possui tamanho fixo, podendo diminuir ou aumentar dependendo da demanda de memória para outras operações. Em momentos de pressão de memória, algumas páginas presentes na *page cache* devem ser liberadas. Pra esse propósito, o *kernel* do Linux utiliza o algoritmo *Least Recently Used* (LRU) com duas listas de páginas: lista ativa e lista inativa. Páginas presentes na lista ativa não são cogitadas para sair da *page cache*, diferentemente das páginas presentes na lista inativa. Quando uma página é trazida para a memória, ela é colocada inicialmente na lista inativa. Caso seja acessada novamente, é movida para a lista ativa. Um algoritmo de balanceamento também move as páginas entre as listas. Essa abordagem tem

como principal objetivo diminuir a interferências de dados que são acessados uma única vez, pois estes estarão na lista inativa, não prejudicando os dados que são constantemente acessados na lista ativa.

Nas primeiras versões do *kernel* Linux, as páginas da *page cache* eram gerenciadas através de uma Tabela *hash* que contia uma lista duplamente encadeada para entradas conflitantes. Contudo, por problemas de desempenho, o gerenciamento passou a ser realizado através de uma *Radix Tree*. Atualmente, uma nova estrutura denominada *Xarray* está sendo utilizada. A utilização desta nova estrutura se deve ao fato de que Wilcox (2018) indicou que a interface provida pela estrutura *Radix Tree* implementada no *kernel* não se comportava como uma árvore, mas sim como um *array* que alterava seu tamanho automaticamente. Por isso, foram propostas modificações na interface, além de melhorias nas politicas de sincronização que tiram vantagens do mecanismo *Read, Copy, Update* (RCU) para realizar buscas paralelas sem gargalo.

# 3 REVISÃO DA LITERATURA

A área de sistemas de arquivos criptográficos tem sido bastante explorada através dos anos. A maioria dos trabalhos buscam compensar a perda de desempenho causada pela criptografia, além de flexibilidade para evitar modificações nas interfaces para as aplicações. Os trabalhos relacionados podem ser classificados de acordo com o contexto de sua execução.

### 3.0.1 Trabalhos implementados em espaço de *kernel*

Zadok et al. (1998) apresentam o CryptFS, um dos sistemas de arquivos criptográficos pioneiros, implementado em espaço de *kernel*. Entre os seus objetivos estava a ideia de criar uma camada transparente de criptografia, sem a necessidade de *daemons* em espaço de usuário. Para alcançar tal objetivo, o CryptFS se baseia na ideia de empilhar uma camada de criptografia acima do sistema de arquivo comum, como mostra a Figura 3.1.

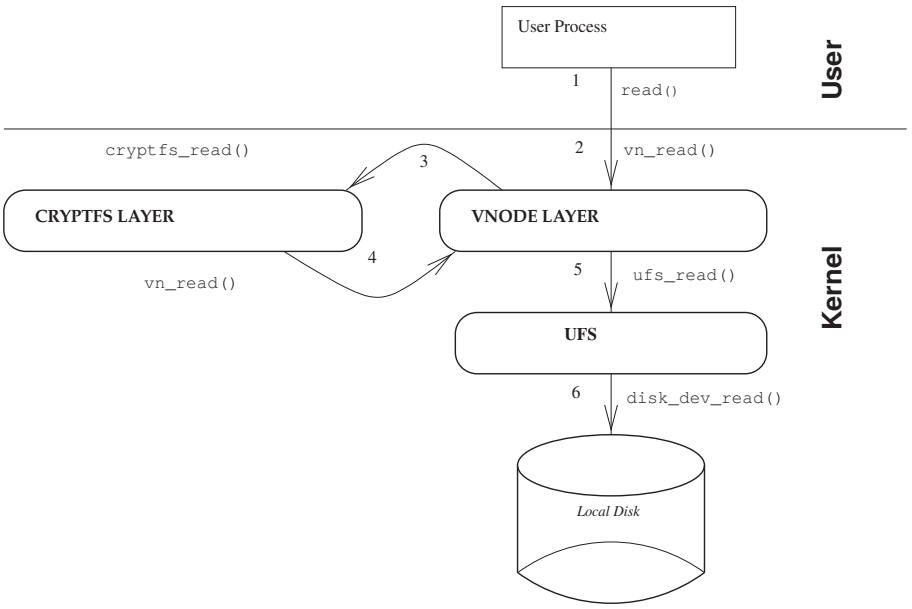



Figura 3.1: Arquitetura do CryptFS (Zadok et al., 1998).

Como a implementação foi realizada em três sistemas operacionais com *kernels* diferentes (Solaris, Linux e FreeBSD), algumas nomenclaturas se diferem, mas a ideia das camadas de software estão presentes em todos eles. Através da camada Vnode/VFS, as chamadas de sistemas importantes podem ser interceptadas de forma transparente, evitando a complexidade da implementação de um sistema de arquivos completo. O sistema de arquivos proposto conta com gerenciamento de chaves, que leva em conta a sessão do usuário, e criptografia Blowfish (Nie e Zhang, 2009) em modo de operação CBC.

Halcrow (2005) apresenta o eCryptFS através do incremento do sistema de arquivos criptográfico proposto por Zadok et al. (1998), agregando funcionalidades que permitem a utilização do sistema de arquivos em ambientes empresariais de grande escala. A arquitetura empilhada permanece a mesma, contudo, a utilização do Linux *Keyring* (Kernel Development Community, 2023b), *Pluggable Authentication Modules* (Levin e Mráz, 2023), OpenSSL (OpenSSL Comitters, 2023), *Trusted Platform Module* (ISO/IEC 11889-1:2015, 2015) e GnuPG

*Keyring* (GNU Project, 2015) incrementam diversos pontos relacionados à segurança e aos requisitos de um sistema de arquivos utilizado em larga escala. Como alguns dos novos recursos são disponíveis apenas para espaço de usuário, o eCryptFS utiliza *daemons* que lidam principalmente com o gerenciamento de chaves, mas mantém as operações criptográficas em *kernel*. O algoritmo de encriptação utilizado é o AES em modo CBC, implementado na Crypto API do *kernel* Linux. Para contornar a dificuldade inerente ao modo de operação CBC, que cria dependências entre os blocos, o eCryptFS quebra o arquivos em estruturas denominadas *extents*, diminuindo a quantidade de dados dependentes um dos outros, como mostra a Figura 3.2. O eCryptFS, que é parte do *kernel* Linux desde a versão 2.6.19, é utilizado como base para o sistema de encriptação do diretório `/home` da distribuição Linux Ubuntu, nativamente pelo ChromeOS e em diversos dispositivos Network Attached Storages (NAS).

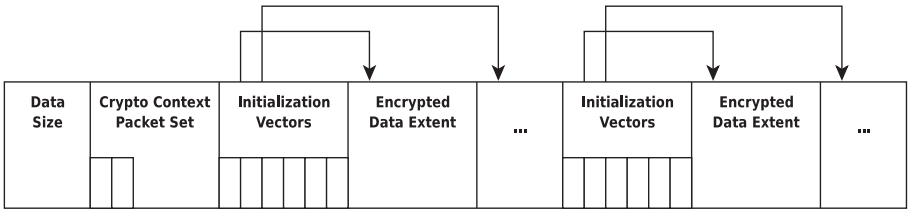



Figura 3.2: Formato de arquivo do eCryptFS (Halcrow, 2005).

Wang et al. (2012) propõe a melhora do sistema de arquivos criptográfico eCryptFS através da otimização da utilização da *page cache*, a utilização de verificação HMAC para identificação de modificações não autorizadas e dados corrompidos, utilização de um *thread pool* para encriptação e escrita no sistema de arquivos base e pequenas otimizações nas operações de escrita. A arquitetura de sistemas de arquivos empilhados utilizada pelo eCryptFS é baseada no VFS. Quando uma chamada de sistema `vfs_read()`, por exemplo, é executada, o VFS repassa a requisição ao eCryptFS. Este, por sua vez, após realizar as operações necessárias, repassa a chamada para o sistema de arquivos que está abaixo dele. Essa operação é realizada através de uma nova chamada à função `vfs_read()`. Contudo, operações realizadas pelo VFS afetam diretamente a *page cache*. Dessa forma, o eCryptFS utiliza o dobro de páginas necessárias para manter um arquivo na memória: páginas encriptadas e páginas não encriptadas. Wang et al. (2012) utiliza o modo de abertura do arquivo para otimizar a utilização de *cache*. Caso o arquivo seja aberto em modo somente leitura, páginas com dados encriptados podem ser liberados imediatamente, da mesma forma que páginas com dados decriptados podem ser liberadas caso o arquivo seja aberto em modo somente escrita.

Xiao et al. (2018) também propõe melhorias para o sistema de arquivos criptográfico eCryptFS, resultando no sistema denominado NV-eCryptFS. O trabalho em questão busca melhorar o fluxo de execução do eCryptFS através da utilização de memórias Non-Volatile Memory (NVM). A utilização de memória persistente permite a alteração da arquitetura do sistema de arquivos pois a *cache* passa a ser não volátil. Por isso, Xiao et al. (2018) apresenta a arquitetura representada na Figura 3.3. O fluxo de operação é encurtado, pois assim que o dado chega ao dispositivo NVM pode ser considerado como persistido. Como a capacidade de armazenamento dos dispositivos NVM são reduzidas, um algoritmo de escalonamento agrega as operações já realizadas com a ajuda da identificação de padrões de acesso, a fim de repassar os dados para os dispositivos de armazenamento mais lentos de forma assíncrona e paralela às operações de encriptação.

Xiao et al. (2020) propõe o Load-aware Adaptative Cache Management (LACM), que tem como objetivo aumentar a eficiência da *cache* para sistemas de arquivos empilhados. Também com base no sistema de arquivos eCryptFS, o LACM propõe duas abordagens para o

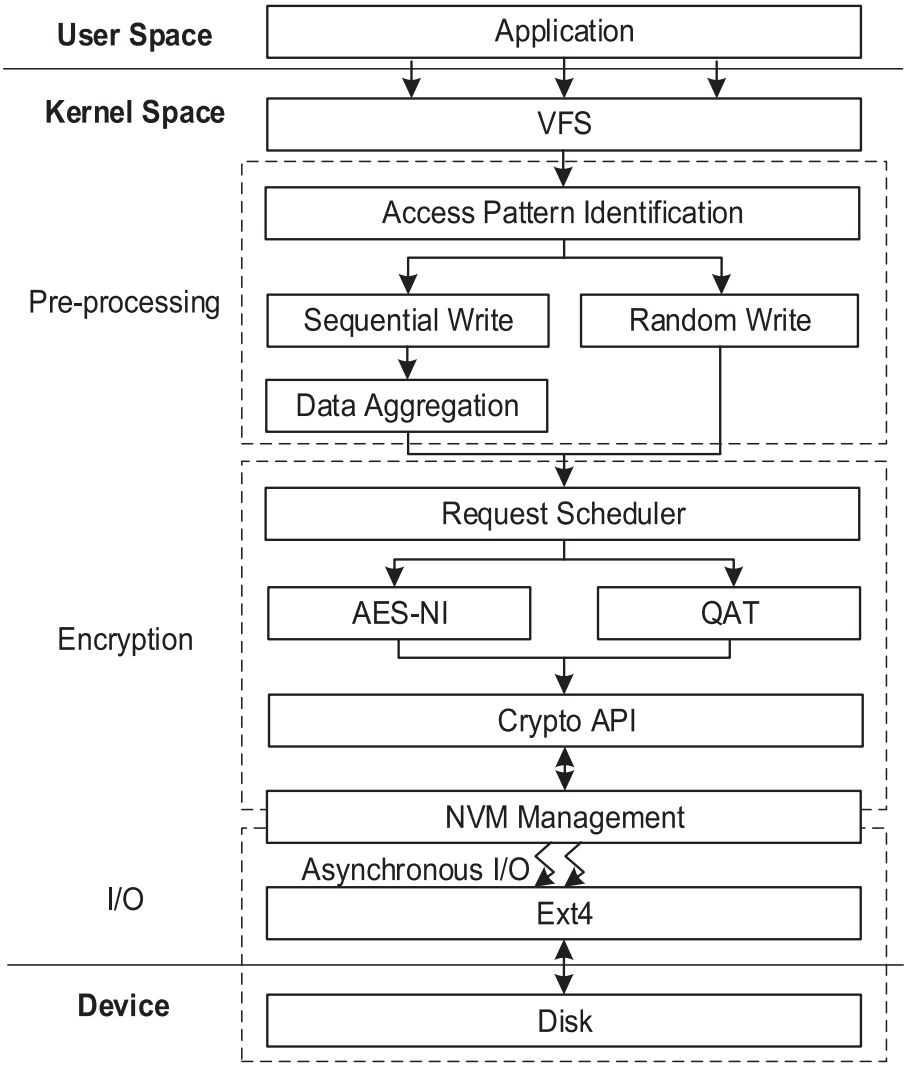



Figura 3.3: Arquitetura do NV-eCryptFS (Xiao et al., 2018).

gerenciamento da *cache*: *Load-aware redundance elimination* e *Non-redundant cache conversion*. Enquanto Wang et al. (2012) tenta eliminar páginas redundantes através do modo que o arquivo foi aberto, Xiao et al. (2020) apresenta uma abordagem mais dinâmica, de forma que a *cache* do sistema de arquivos criptográficos nunca possui a mesma página mapeada pela *cache* do sistema de arquivos inferior. Para garantir a não redundância de forma eficiente, a característica *Load-aware* realiza papel importante, descartando páginas de texto claro após as operações de escrita, assim como descartando páginas encriptadas após operações de leitura. Contudo, existem cenários em que uma página presente em uma *cache* tenha que ser movida para outra. Caso uma página presenta na *cache* decriptada seja escrita, os dados serão encriptados e repassados para a *cache* encriptada, enquanto a primeira será descartada imediatamente. Porém, quando uma página presente na *cache* encriptada é lida, esta não é descartada, pois isso implicaria em uma operação de E/S síncrona. Portanto, o dado é decriptado e retornado, mas não permanece na *cache* de paginas decriptadas.

Eckert et al. (2000) propõe o GSFS, um sistema de arquivos criptográfico que tem como principal objetivo dar suporte a um sistema de arquivos multi-usuário, através controles de acesso baseados em grupos, além do comportamento dinâmico entre os integrantes dos grupos. Para isso, foi decidido que cada grupo tem um *owner*, que é capaz de gerenciar as permissões que são presentes em Access Control Lists (ACLs). GSFS é um dos únicos sistemas de arquivos

que suportam criptografia assimétrica, mesmo não provendo bom desempenho. Assim como eCryptFS, algumas funcionalidades do sistema de arquivos são providas em espaço de usuário, por questões de acesso às interfaces.

Sharma et al. (2006) apresenta o TransCrypt, um sistema de arquivos criptográfico que prioriza mitigar problemas de segurança em detrimento ao desempenho. Sua arquitetura é dirigida por um *threat model* que leva em consideração tanto ataques offline quanto ataques online. Algumas especificações são colocadas pelo Transcrypt como integridade de arquivos, agentes de recuperação, confiança no mínimo de entidades, entre outros. A sua implementação é realizada a partir de *hooks* no sistema de arquivos, além da utilização *smart cards* em *user-space* para o gerenciamento de chaves e autenticação de usuários.

Lee et al. (2007) propõe a utilização dos mecanismos de encriptação no nível de cache de disco, com o objetivo de aplicar a criptografia em sistemas embarcados. Esse tipo de sistema tem CPUs com velocidades limitadas, pouca memória e baterias, de forma que os efeitos colaterais da aplicação da criptografia são mais evidentes. Quando os dados são lidos do dispositivo de armazenamento, normalmente são trazidos mais dados que o requisitado, seja pelo fato da leitura em blocos ou pela aplicação de algum algoritmo que analisa os padrões de acesso. Se a criptografia for aplicada no nível do *driver*, todos esses dados passariam por operações criptográficas, inclusive os dados que não foram explicitamente requisitados. Uma implementação empilhada não permite o acesso ao *buffer*, ou seja, se a aplicação buscasse o mesmo dado duas vezes, as operações criptográficas se repetiriam duas vezes. Portando, essa proposta busca realizar as operações criptográficas na própria cache do disco, mas levando em consideração o dado que foi explicitamente requisitado. Dessa maneira, a cache pode conter tanto dados encriptados quanto dados decriptados, mas a quantidade de operações criptográficas realizadas é menor, influenciando diretamente o desempenho em sistemas embarcados.

Rawat e Kumar (2012) propõe o ECFS, um sistema de arquivos criptográfico em espaço de *kernel* empilhado, que disponibiliza criptografia de blocos AES através do modo de operação XTS, além de autenticação de usuários e compartilhamento de arquivos por meio criptografia assimétrica. ECFS também utiliza ACL para permissões de acesso e *extended attributes* para armazenamento de metadados. Um *daemon* em espaço de usuário é utilizado como um canal para transferência de pacotes entre o *kernel* e o armazenamento de chaves.

Trusted Platform Module (TPM) (ISO/IEC 11889-1:2015, 2015) é um hardware capaz de gerar e armazenar chaves criptográficas. Diversos trabalhos que lidam com criptografia utilizam deste recurso para prover mais segurança para suas aplicações através da API disponibilizada em *user-space*. Além dos sistemas de arquivos criptográficos já citados que utilizam este módulo, Jun e Huiping (2010); Jin e Bo (2011) também basearam seus trabalhos neste componente, aplicando-o na criptografia em nível de sistemas de bloco e em nível de sistemas de arquivos empilhados, respectivamente. Ambos utilizam módulos implementados em espaço de usuário para realizar a comunicação com o TPM e realizar gerenciamento de chaves.

Dm-Crypt (Kernel Development Community, 2023a; Broz, 2023a) é um *device mapper*, ou seja, mapeia um dispositivo físico (encriptado) em um dispositivo virtual (decriptado). Esse sistema utiliza a biblioteca Crypto API para realizar as operações criptográficas durante a transferência de dados entre o dispositivo de armazenamento e a memória. Essa abordagem é comumente utilizada na encriptação de todo do dispositivo de armazenamento, sem distinção entre dados e metadados, granularidade de arquivos e usuários. Atualmente, o Dm-crypt é amplamente utilizado juntamente com o Linux Unified Key Setup (LUKS) (Broz, 2023b).

### 3.0.2 Trabalhos implementados em espaço de usuário

Os trabalhos citados que implementam sistemas de arquivos em espaço de usuário utilizam a biblioteca FUSE (Szeredi, 2003). Esta biblioteca disponibiliza uma interface de comunicação com o módulo FUSE, que está no *kernel* Linux. Quando uma chamada de sistema é realizada, o VFS repassa a chamada para o módulo FUSE em *kernel*, como se este fosse o sistema de arquivos responsável por tratar a requisição. Contudo, essa chamada é repassada para a biblioteca FUSE em espaço de usuário, a qual repassa a requisição para a implementação do sistema de arquivos também em espaço de usuário.

Sharma et al. (2012) propõe o UsiFE, um sistema de arquivos criptográfico que tem como principal característica diminuir a granularidade da encriptação, podendo encriptar somente partes importantes de arquivos, diminuindo o tempo de processamento. A ideia central é de que arquivos como XML não precisam ser encriptados completamente, ou que alguns usuários podem ter acessos somente a determinadas partes de arquivos. Por esse motivo, o UsiFE implementa duas formas de visualização: *display* e *stealth*. Na forma *display*, o UsiFE não tenta decriptar partes dos arquivos que o usuário não possui a chave válida para decriptá-las, mas exibe todo o arquivo. Na forma *stealth*, o USiFE só disponibiliza as partes que ele pode decriptar.

Pontes et al. (2017) apresenta o SafeFS, uma arquitetura modular para sistemas de arquivos criptográficos em espaço de usuário. Tendo em vista que algumas políticas de privacidade, confidencialidade e integridade podem mudar com o tempo, assim como a granularidade dos dados tratados pelo sistema de arquivos, camadas de software podem ser adicionas, retiradas ou substituídas para que o sistema se adapte ao cenário desejado. Para resolver estes problemas, a arquitetura demonstrada na Figura 3.4 foi proposta. A ideia central do trabalho é permitir que o SAC seja composto por diversas camadas de software independentes, proporcionando alta flexibilidade na adaptação do sistema. O trabalho também demonstrou a sua eficácia através da implementação de uma camada que lida com a granularidade de dados, uma camada de encriptação e uma camada que trata a persistência dos dados. Além dessa implementação, Esteves et al. (2019) utiliza a arquitetura do SafeFS para implementar um sistema de arquivos criptográfico em espaço de usuário com o auxilio em hardware do SGX, objetivando a sua execução em ambientes não confiáveis.

Gough (2020) apresenta o EncFS, um sistema de arquivos criptográficos em espaço de usuário similar ao eCryptFS, que provê encriptação transparente, utilizando implementações presentes na OpenSSL. Dentre as suas funcionalidades, EncFS não necessita de permissões de *root* para sua execução, tendo em vista que sua arquitetura é empilhada sobre um sistema de arquivos base através da biblioteca FUSE. Em decorrência da sua popularidade, alguns trabalhos como Leibenger et al. (2016) buscam incrementar funcionalidades ao EncFS como o suporte a vários usuários através de políticas de controle de acesso.

O EncFS++, proposto por Eduardo et al. (2019); Eduardo (2019) é o SAC que mais influencia este trabalho. Implementado com base no EncFS, o Encfs++ utiliza a WAESlib para utilizar criptografia especulativa e processamento paralelo acelerado pela GPU. Além da utilização desse tipo de criptografia, o trabalho em questão apresenta formas de geração, armazenamento e gerenciamento de *nonces*, assim como o gerenciamento dos contextos criptográficos da WAESlib. Como resultado, o EncFS++ apresentou diminuição da latência das operações, aumento da vazão de dados e melhor utilização da CPU. Detalhes de design e implementação estão descritos na Seção 4.1.

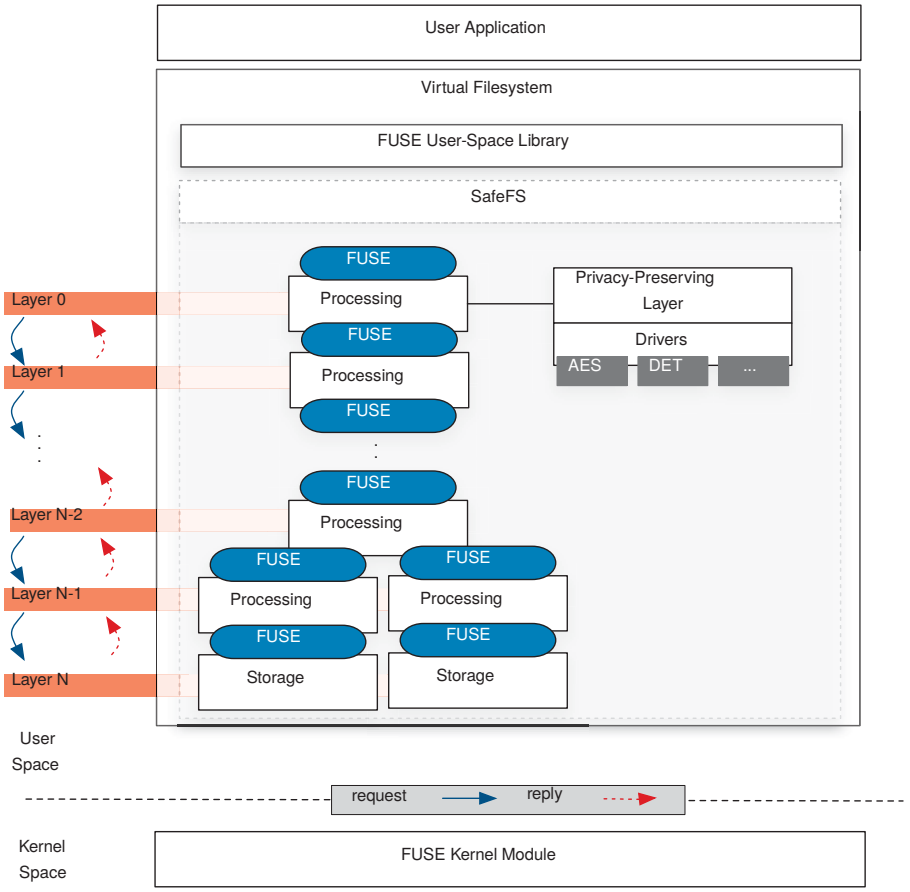



Figura 3.4: Arquitetura do SafeFS (Pontes et al., 2017).

## 3.1 CONSIDERAÇÕES FINAIS

Após o levantamento dos trabalhos relacionados, fica perceptível a importância e a aplicabilidade de sistemas de arquivos criptográficos no contexto atual da segurança da informação. Os trabalhos presentes na literatura indicam uma preferência por utilização de arquiteturas empilhadas, tendo em vista a facilidade de implementação e reutilização de funcionalidades já providas pelos sistemas de arquivos existentes. Contudo, o custo inerente ao grande número de trocas de contexto e necessidade de um gerenciamento de *cache* adicional faz com que o desempenho dos sistemas atuais seja prejudicado (Vangoor et al., 2017).

É perceptível também que existem soluções que realizam as operações criptográficas em nível de *driver*, em nível requisições dos usuários e até mesmo em um ponto intermediário entre eles, em cache do dispositivo de armazenamento. Contudo, essas abordagens não são aplicadas em conjunto e as vantagens que cada nível de execução ainda não foram exploradas em uma única solução.

# 4 PROJETO E IMPLEMENTAÇÃO

A ideia central deste trabalho é que cada camada do subsistema de armazenamento do Linux tem potencial para contribuir com a criptografia antecipada e especulativa. Para que seja possível a aplicação desta abordagem, se faz necessária uma execução dentro do *kernel* e utilização de *hooks* em diferentes pontos da pilha de armazenamento. Uma prova de conceito foi implementada para demonstrar o desempenho da arquitetura proposta e permitir a comparação com o estado da arte. O sistema implementado, baseado no sistema de arquivos EXT4, foi denominado EXT4james.

A Seção 4.1 apresenta as principais limitações do EncFS++ e que constituem desafios para esta proposta; a Seção 4.2 explora a viabilidade e os potenciais desta proposta; a Seção 4.3 descreve a arquitetura utilizada para realizar o armazenamento e gerenciamento dos metadados criptográficos; a Seção 4.4 apresenta os novos fluxos de execução criados para lidar com a criptografia; a Seção 4.5 descreve como a WAESlib foi adaptada para executar em *kernel* e como foi realizado o gerenciamento dos seus contextos criptográficos.

## 4.1 LIMITAÇÕES DO ENCFS++ E DESAFIOS DE OTIMIZAÇÃO

Eduardo et al. (2019) demonstrou o potencial da utilização da criptografia especulativa em sistemas de arquivos através do EncFS++. Contudo, a execução em nível de usuário e a arquitetura empilhada implicam em restrições no seu design e perdas de desempenho. Afim de exemplificar essas dificuldades encontradas pelo EncFS++, esta seção analisa alguns conceitos da sua arquitetura.

Figura 4.1 apresenta o fluxo de execução simplificado do EncFS++. O primeiro ponto a se considerar é a grande quantidade de trocas de contexto inerentes à execução em espaço de usuário. O módulo FUSE se registra no VFS como um sistema de arquivos, de forma que requisições vindas de aplicações são repassadas a ele, que por sua vez repassa para implementação em espaço de usuário, que por sua vez repassa para um sistema de arquivos base. O segundo ponto está relacionado ao momento que as operações criptográficas são realizadas. Ainda que esteja fora do escopo do EncFS++ manter dados em memória encriptados, ele herda do EncFS (SAC no qual o EncFS++ foi baseado) a aplicação das operações criptográficas no nível de requisições do usuário. Essa restrição imposta pela arquitetura empilhada impossibilita a realização das operações criptográficas em um nível mais baixo e por isso gera trabalhos desnecessários de encriptação e decriptação de dados em memória.

Decorrente da utilização do modo de operação CTR, existe um *nonce* de 16 bytes associado a cada bloco de dados do EncFS++ que é utilizado como parâmetro de entrada para a criação da máscara de decriptação. Tendo em vista que o desempenho geral do sistema depende da forma com que esse metadado é armazenado e acessado, o EncFS++ propôs uma estrutura denominada *Nonce Node* (`N-node`), apresentado na Figura 4.2b. Cada arquivo de dados possui um `N-node`, o qual armazena 16 *nonces* que mapeiam os primeiros 16 blocos de dados. Caso o tamanho do arquivo de dados ultrapasse 16 blocos (64KB para blocos de 4096 bytes), é criado um arquivo dedicado para armazenar os *nonces* remanescentes, como mostra a Figura 4.2c. Como apresentado na Figura 4.2a, o EncFS++ propõe a utilização de um arquivo global que agrupa todos os `N-nodes` presentes no sistema de arquivos, com o objetivo de gerenciá-los.

O gerenciamento do arquivo global é realizado através de um *bitmap* que contém 524128 bits. Isso indica que o número total de arquivos que podem existir no sistema de arquivos é

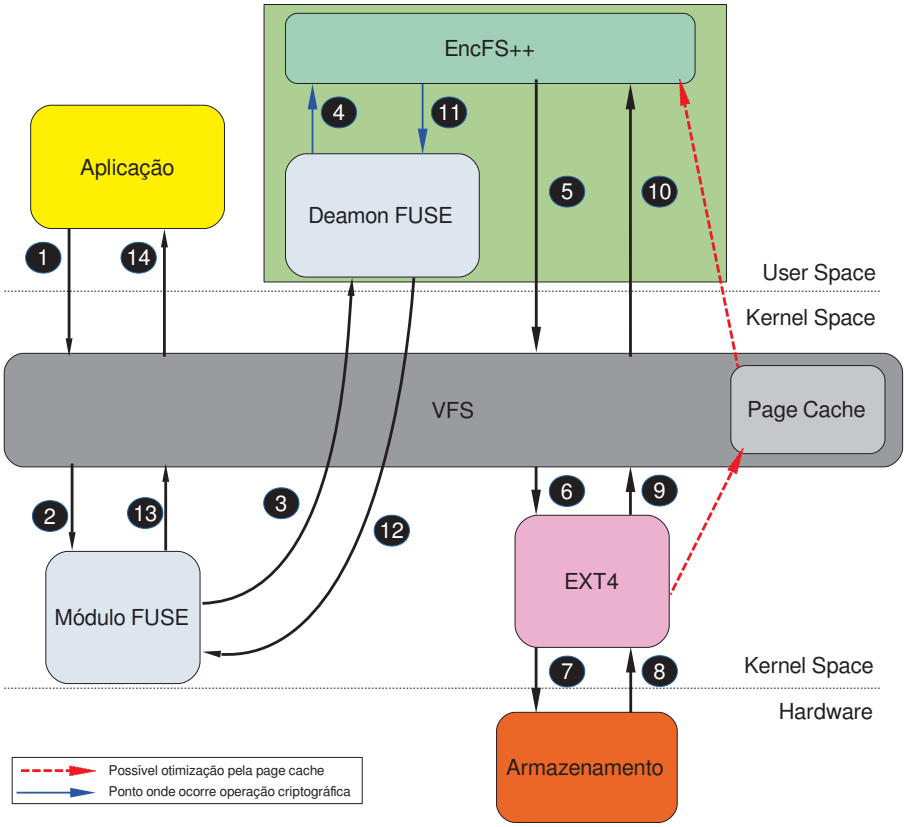



Figura 4.1: Fluxo simplificado de uma operação de leitura ou escrita do EncFS++.

524128, o que é muito pouco já que o EXT4 permite a existência de $2^{32}$ arquivos, por exemplo. Uma solução ingênua seria simplesmente aumentar o número do *bitmap* e manter a arquitetura do arquivo global, visto que o *overhead* da iteração pelo *bitmap* se tornaria muito grande. Por isso, no presente trabalho, foi proposto um bitmap hierárquico baseado no gerenciamento de volumes do EXT4.

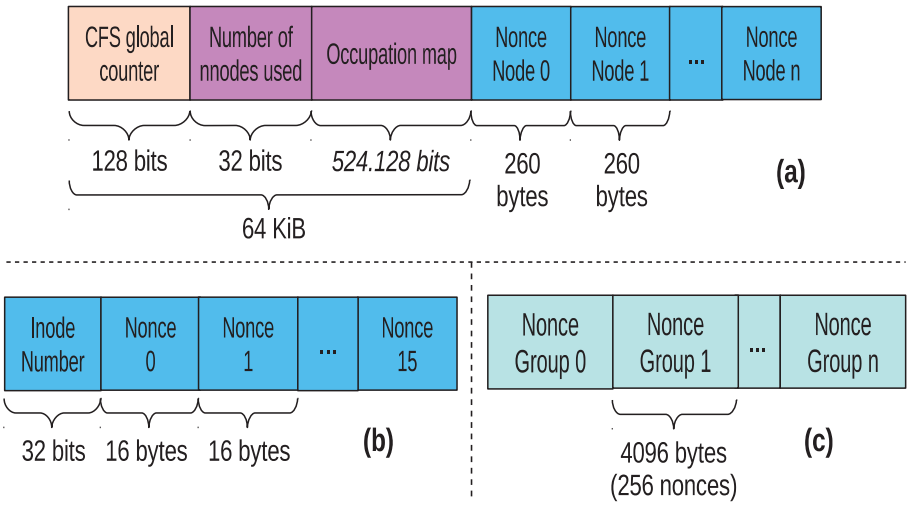



Figura 4.2: (a) Formato do arquivo global. (b) *Nonce node* (*N-node*). (c) Formato do arquivo dedicado de *nonces*. (Eduardo et al., 2019).

A WAESlib disponibiliza contextos criptográficos que são utilizados para realizar a requisição de criação de uma máscara e a sua aplicação. O EncFS++ gerencia esses

contextos criptográficos através de janelas deslizantes, como a Figura 4.3. Para as operações de escrita, uma janela deslizante global foi criada, afim de manter máscaras criptográficas criadas especulativamente com *nonces* ainda não utilizados, possibilitando a aplicação dessas máscaras em qualquer operação de escrita no sistema de arquivos. Para as operações de leitura, o EncFS++ mantém duas janelas deslizantes por arquivo aberto. No momento de abertura do arquivo, o Encfs++ preenche as duas janelas com as máscaras criptográficas que mapeiam os primeiros blocos de dados do arquivo: uma janela que contém máscaras que serão utilizadas em leituras sequenciais e a outra para leituras randômicas. Quando o bloco de dados 0 for lido em uma leitura sequencial, por exemplo, a máscara do primeiro contexto é aplicada sobre o dado lido e logo em seguida, o contexto em questão inicia o preparo da máscara de encriptação do próximo bloco ainda não mapeado, realizando assim uma utilização cíclica dos contextos. Tendo em vista que esta nova máscara requisitada só será utilizada após a aplicação de todas as outras máscaras presentes na janela, a sua criação é realizada em baixa prioridade, permitindo que máscaras mais urgentes, como de leituras, tenham preferência.

A necessidade da existência de uma janela para leitura sequenciais e outra para leituras randômicas se dá pelo fato de que as máscaras são criadas especulativamente, ou seja, a leitura de um bloco fora da sequência pode invalidar máscaras já criadas. A cada leitura randômica, todas as máscaras da janela devem ser recalculadas para que seja possível decriptar os dados em questão. Por esse motivo, a janela de máscaras referente aos acessos randômicos é menor que a janela utilizada nas leituras sequenciais, objetivando a diminuição das invalidações. Contudo, como mostrado na Seção 4.2, o tempo de E/S dos dispositivos de armazenamento são maiores que os necessários, o que indica que a criação das máscaras utilizadas nas operações de leitura não precisa ser especulativa e possui um potencial de otimização.

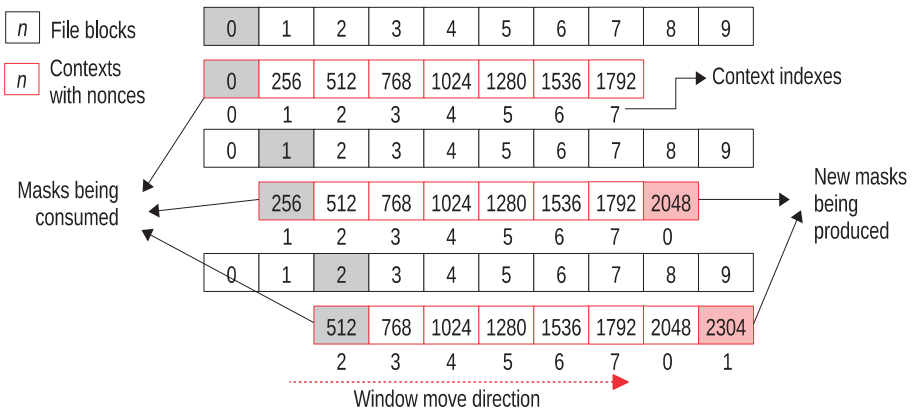



Figura 4.3: Gerenciamento de contextos criptográficos no Encfs++ (Eduardo et al., 2019).

## 4.2 VIABILIDADE E POTENCIAIS

A Figura 4.4a apresenta a execução de uma requisição de leitura de um SAC convencional. Mesmo que a CPU esteja livre para executar algum trabalho enquanto a operação de E/S está sendo tratada pelo dispositivo de armazenamento, todas as operações são realizadas de forma sequencial porque os dados precisam estar em memória para que possam ser processados. Por outro lado, através da aplicação da criptografia especulativa mostrada na Figura 4.4b, a dependência de dados é restringida somente ao passo em que ocorre a execução do XOR, de forma que o tempo ocioso da CPU previamente desperdiçado é utilizado para a criação de máscaras criptográficas, escondendo grande parte da latência introduzida pelas operações de decriptação.

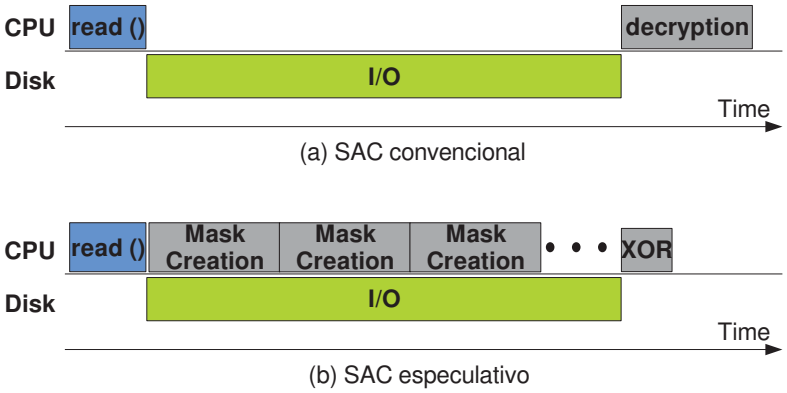



Figura 4.4: Execução da operação de leitura de um SAC convencional.

Quando o SAC realiza uma operação de leitura, uma requisição de criação de máscaras é submetida à WAESlib e as *threads* trabalhadoras realizarão o trabalho. Quando o dado lido e a máscara estiverem prontos, a decriptação do dado é finalizada por meio da aplicação do XOR entre eles.

É razoável pensar que quanto maior o tempo da operação de E/S mais máscaras podem ser criadas, gerando maior desempenho. De fato, operações de E/S longas permitem a criação de muitas máscaras. Contudo, a utilização de dispositivos de armazenamento de alta latência como HDDs implica em um impacto insignificante das operações criptográficas, tendo em vista que o tempo de E/S representa quase 100% do tempo total da requisição (Zhong et al., 2022). Isso esconderia as otimizações das operações criptográficas. Contudo, utilizar dispositivos de baixa latência faz com que o impacto das operações de software chegue próximo de 50%, o que no contexto de sistemas de arquivos criptográficos indica um maior impacto das operações criptográficas. Dessa forma, as otimizações propostas neste trabalho apresentam maior potencial de otimização.

Neste cenário, duas questões surgem: primeiramente, para um dispositivo de baixa latência em que o *overhead* de software é significativo, existe espaço suficiente para criar máscaras criptográficas utilizando um processador *multicore*? Essa pergunta pode ser respondida analisando a Tabela 4.1 que mostra os tempos de uma requisição de leitura, iniciando logo após a submissão para a camada de E/S de blocos e finalizando na interrupção gerada pelo dispositivo logo após a finalização da operação. Um bloco de dados de 4096 bytes foi requisitado para um dispositivo NVMe e para um dispositivo em RAM. A tabela 4.2 mostra os tempos para a encriptação de um bloco de 4096 bytes e o tempo para realizar o XOR entre dois blocos de 4096 bytes.

Analisando os resultados, é perceptível que o dispositivo NVMe provê muito mais espaço que o necessário para criação de máscaras, e o dispositivo baseado em RAM provê menos espaço que o necessário. A segunda pergunta que surge é a seguinte: para dispositivos de armazenamento de baixas latências como a memória RAM, onde o tempo de E/S é menor que o tempo de criação de máscaras, é possível obter melhoras de desempenho quando comparados com SACs convencionais? Como apresentado no Capítulo 5, sim: quanto mais espaço de E/S for aproveitado durante as requisições, mais otimizada será a operação, ainda que não seja possível finalizar todo o processamento neste tempo, visto que todo o tempo ocioso da CPU será aproveitado. Essa característica faz com que esta proposta gere melhores resultados a medida que os dispositivos fiquem mais rápidos, possibilitando o acompanhamento da evolução tecnológica na área de armazenamento.

Tabela 4.1: Tempo de E/S para ler 4096 bytes de um sistema de arquivos armazenado em RAM e em NVMe. A requisição foi realizada utilizando a flag O_DIRECT. A RAM utilizada possui 16GB, 2933MHz, DDR4. O NVMe utilizado foi um ADATA SX6000LNP.

| Backed Device | I/O time |
|---|---|
| RAM | 557 ns |
| NVMe | 18456 ns |

Tabela 4.2: Tempo gasto para realizar o XOR entre dois blocos de 4096 bytes e para criar uma máscara criptográfica de 4096 bytes. O XOR foi realizado utilizando instruções AVX e as máscaras foram criadas utilizando o a Linux Kernel Crypto API. O processador utilizado foi um Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz, com suporte AES-NI habilitado.

| Operation | Time |
|---|---|
| XOR | 318 ns |
| Mask Creation | 1192 ns |

Máscaras criptográficas criadas especulativamente para as operações de leitura podem ser invalidadas em caso de um acesso não sequencial. Contudo, a partir da análise demonstrada anteriormente, percebe-se que a criação especulativa não é necessária tanto para leituras sequenciais quanto para leituras randômicas, pois o tempo de E/S é suficiente para criar máscaras que estão envolvidas na operação em questão, evitando assim a ocupação da CPU com a criação de máscaras que podem ser invalidadas.

Se para operações de leitura as máscaras podem ser criadas *ahead-of-time*, durante a operação de E/S, máscaras criptográficas para operações de escrita podem ser criadas ainda antes, de forma especulativa, tendo em vista que não existe a necessidade de buscar por *nonces* e as únicas máscaras invalidadas serão as criadas logo antes do sistema de arquivos ser desmontado.

Para que essas funcionalidades sejam implementadas da melhor forma, todas as camadas do subsistema de armazenamento do *kernel* Linux podem contribuir: a camada do VFS pode ser utilizada para realizar o armazenamento dos *nonces* de forma simples e transparente, possibilitando até a utilização de diferentes dispositivos de armazenamento para armazenar estes metadados; a camada de sistema de arquivo pode prover a granularidade de arquivos e funcionalidades relacionadas a arquivos como *extented attributes*, permitindo a encriptação por arquivos utilizando diferentes chaves e suporte para sistemas multi-usuário; a camada de E/S de blocos é a que realiza as requisições para o dispositivo de armazenamento, tornando-a um bom ponto para submeter as criações de máscaras; as funções chamada pelo dispositivo de armazenamento através de interrupções que sinalizam o fim das operações de E/S podem ser utilizadas para disparar a aplicação das máscaras criptográficas.

A forma de aplicação da criptografia antecipada (*ahead-of-time*) e especulativa proposta neste trabalho tem potencial de gerar ganhos de desempenho em arquiteturas de trabalhos recentes que modificam a pilha de armazenamento padrão do Linux. O trabalho proposto por Ruan et al. (2019), por exemplo, utiliza computação *in-storage*, enquanto o trabalho proposto por Hwang et al. (2021) trata de forma diferente aplicações sensíveis à latência e aplicações sensíveis à vazão. Por outro lado, diversos trabalhos (Zhong et al., 2022; Zhang et al., 2019; Yang et al., 2017) realizam o *bypass* da pilha de armazenamento do *kernel* e modificam completamente o fluxo de execução de requisições de E/S.

Computação *in-storage* pode permitir a aplicação das máscaras em nível de hardware, diminuindo a troca entre as *threads* e a carga da CPU, possivelmente aumentando o desempenho geral do sistema. O tratamento diferenciado dos tipos de requisições pode tirar vantagem da prioridade de contextos para gerenciar a ordem de criação das máscaras. Arquiteturas realizam o

*bypass* da pilha de armazenamento do *kernel* ainda precisam realizar requisições ao dispositivo de armazenamento. Esses pontos podem ser utilizados para disparar requisições de máscaras criptográficas, independentemente da execução estar sendo feita em espaço de usuário ou espaço de *kernel*. Além disso, estes trabalhos buscam diminuir o tempo de execução do *kernel*, o que tornaria o tempo das operações criptográficas mais relevantes na latência total das operações, permitindo maiores otimizações pela proposta deste trabalho.

O modo de operação CTR possui a desvantagem de necessitar de um *nonce* para cada bloco de 128 bits. Contudo, como proposto por Eduardo et al. (2019), é possível utilizar um *nonce* randômico inicial para cada bloco de dados (4096 bytes) e produzir o seu sucessor numérico para cada bloco de 128 bits subsequente do bloco de dados. Dessa forma, o *overhead* de armazenamento gerado pelos *nonces* não é relevante quando comparado com o espaço total: em um bloco de 4096 bytes, é possível armazenar *nonces* suficientes para mapear 1MB de dados, mantendo o *overhead* abaixo de 0.4%.

Para validar a proposta, foi implementada uma prova de conceito utilizando o sistema de arquivos EXT4 como subsistema central, tendo em vista a granularidade de arquivos e a interface tanto com o VFS quanto com a camada de E/S de blocos. Contudo, alguns *hooks* também foram inseridos em diferentes pontos da pilha de armazenamento do *kernel* Linux. O novo SAC foi denominado EXT4james.

Este trabalho não visa a segurança dos SACs, mas sim o desempenho. Dessa maneira, o modelo de ameaça do sistema proposto se assemelha ao do fscrypt (Kernel Development Community, 2023f), um SAC também implementado em espaço de *kernel* através de *hooks*. Vale ressaltar que o gerenciamento de chaves está fora o escopo deste trabalho, contudo, esse tema é amplamente explorado na literatura e diferentes soluções podem ser utilizadas em conjunto com esta proposta. Mesmo que o sistema proposto não encripte os metadados dos arquivos de dados, a confidencialidade dos dados é garantida para ataques *offline*. A encriptação de dados em memória está fora do escopo deste trabalho, de forma que ataques *online* como *side-channel* e comprometimento do espaço de memória do *kernel* não foram levadas em consideração. A implementação proposta utiliza a Crypto API (Mueller e Vasut, 2023), de forma que vulnerabilidades presentes nesta biblioteca podem comprometer o SAC proposto.

## 4.3 ARMAZENAMENTO E GERENCIAMENTO DOS METADADOS CRIPTOGRÁFICOS

Visto que é preciso armazenar um *nonce* de 16 bytes para cada bloco de dados, se faz necessário armazená-los de maneira que o acesso a eles seja eficiente. Como demonstrado na Seção 4.1, é possível otimizar o esquema de gerenciamento do EncFS++. Armazenar os *nonces* juntamente com os dados causaria um desalinhamento dos blocos, gerando desvantagens para o sistema de cache. Outro desafio a se considerar é que armazenar os *nonces* separados uns do outros impossibilita a leitura agregada, gerando acessos desnecessários ao disco.

Para superar os desafios relacionados ao armazenamento dos *nonces*, foi adotado um *design* inspirado no gerenciamento de volumes do EXT4 (Kernel Development Community, 2023d) e no `N-node` utilizado no EncFS++, tirando vantagem da camada do subsistema de armazenamento VFS. Os primeiros 256 *nonces* (totalizando 4096 bytes) dos arquivos de dados serão armazenados em um único arquivo denominado *Global File*. Caso um arquivo de dados ocupe mais que 1MB (necessite de mais que 256 *nonces*), um arquivo dedicado denominado *nonce file* será criado para armazenar os demais metadados do arquivo de dados em questão. Ambos, *Global File* e *nonce files*, não possuem qualquer permissão de escrita, leitura ou execução para qualquer usuário, inclusive *root*, tendo em vista que eles são acessados somente pelo *kernel*.

Buscando tirar total proveito do tamanho máximo de arquivos que o EXT4 disponibiliza, a organização do *Global File* foi feita como mostra a Figura 4.5.

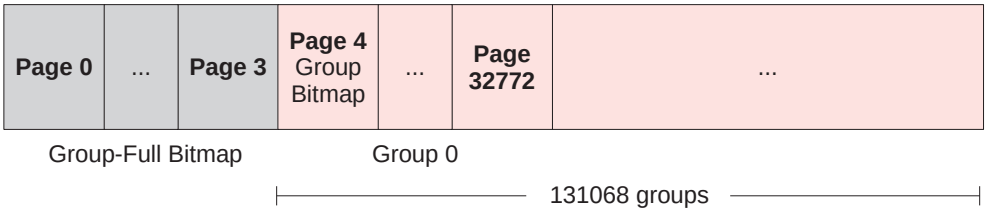



Figura 4.5: Gerenciamento do *Global File*.

Os primeiros 256 *nonces* de um arquivo de dados são armazenados em uma página dentro do *Global File*. Este, por sua vez, é dividido em grupos de páginas. Cada grupo possui $2^{15}$ + 1 páginas, em que a primeira página é um *bitmap* que controla a utilização das outras páginas desse grupo. Todas as outras páginas do grupo armazenam os primeiros *nonces* de diferentes arquivos.

As primeiras 4 páginas do *Global File* compõem o que foi chamado *Group-Full Bitmap*. Esse *bitmap* mapeia os grupos que estão totalmente ocupados e não tem mais espaço para mapear novos arquivos de dados. Esse design possui duas vantagens: bom aproveitamento do tamanho do *Global File*; os *bitmaps* hierárquicos permitem alocações otimizadas durante o processo de criação de arquivos de dados. Para endereçar uma página dentro do *Global File*, um inteiro de 32 bits é utilizado: os primeiros 17 bits endereçam o grupo e os últimos 15 bits endereçam a página dentro do grupo. Essa arquitetura, de fato, reduz a quantidade total de *inodes* disponibilizados pelo EXT4 em 0.003%, pois as páginas de *bitmap* inviabilizam a utilização de 131072 grupos de blocos. Contudo, estes inodes "desperdiçados" podem ser utilizados como *nonce files*.

Tendo em vista que a estrutura `inode` em memória armazena metadados dos arquivos, foi adicionado à ela um ponteiro para uma estrutura que armazena os metadados do SAC proposto. A nova estrutura, denominada `james_inodeinfo`, é apresentada no Código 4.1. Os membros dessa estrutura são descritos nos momentos oportunos.

Listing 4.1: Estrutura james_inodeinfo.

```
1  struct james_inodeinfo
2  {
3     strict james_ivcache *ivcache;
4     struct rw_semaphore ivcache_lock;
5     struct inode *inode;
6     uint64_t global_ivblock;
7     uint64_t iv_ino;
8     struct file *iv_filp;
9     struct file *dio_iv_filp;
10    struct rw_semaphore ivfile_lock;
11    atomic_t refcount;
12    atomic_t blkwrite;
13    atomic_t ivcount;
14    uint8_t unlink_called;
15    void (*james_on_dio_ptr) (struct bio **src_bio,
16                        struct inode *inode,
17                        loff_t pos, loff_t len);
18 };
```

Para relacionar o endereço da página dentro do *Global File* e o *nonce file* (caso exista) com um arquivo de dados, foi utilizada uma funcionalidade provida pela camada de sistema

de arquivos. O EXT4 provê um mecanismo chamado *extended attributes*, através das funções `ext4_xattr_set` e `ext4_xattr_get`, que permite armazenar metadados adicionais no *inode*. Como só é necessário armazenar essas duas informações, é possível utilizar o *inline extended attributes*, que armazena os dados de fato dentro da estrutura *inode* e dispensa a alocação de um bloco dedicado.

No momento da primeira abertura de um arquivo de dados, uma estrutura `james_inodeinfo` é alocada e os *extended attributes* são lidos para completar os membros `global_ivblock` e `iv_ino`, que indicam a página alocada para ele no *Global File* e o *nonce file*, respectivamente.

Para realizar a leitura e escrita no *Global File* e nos *nonce files*, foram utilizadas as interfaces `kernel_read` e `kernel_write` providas pelo VFS. Contudo, a simples utilização da interface não é suficiente, pois o fluxo normal de leitura e escrita passam pelos *hooks* que realizam operações criptográficas, que não devem ser realizadas sobre os metadados. *Hooks* inseridos no início dessas funções verificam se o `inode` do arquivo em questão referencia uma estrutura `james_inodeinfo`. Se o ponteiro for nulo, o arquivo em questão não é um arquivo de dados, então as operações de criptografia não são executadas.

Utilizar a camada de VFS para realizar as operações de leitura e escrita dos *nonces* faz com que as requisições percorram toda a pilha de armazenamento. Para evitar esse *overhead* todas as vezes que um bloco de dados é acessado, um bloco de *nonces* é sempre mantido em memória para cada instância de um arquivo aberto. A a leitura e escrita dos *nonces* são realizadas em granularidade de bloco. O membro *ivcache* da estrutura *james_inodeinfo* é responsável por referenciar a estrutura que armazena e gerencia o bloco de *nonces* em memória e é protegido pelo membro `ivcache_lock` contra acessos paralelos. O Código 4.2 apresenta a estrutura em questão.

Listing 4.2: Estrutura de cache de nonces.

```
1 struct james_ivcache
2 {
3    __uint128_t *ivs;
4    uint32_t block_num;
5    uint8_t flags;
6 };
```

O ponteiro denominado `ivs` aponta para o endereço de memória que contém os metadados; o membro `block_num` indica qual bloco de *nonces* está mapeado no momento; o membro `flags` indica se o bloco de *nonces* foi marcado (está sujo, *dirty*) e precisa ser escrito no dispositivo de armazenamento e se o bloco em questão está presente em memória.

As interfaces `kernel_read` e `kernel_write` realizam operações *buffered I/O*, ou seja, utilizam a *page cache* com o objetivo de diminuir a quantidade de acessos ao dispositivo de armazenamento. Como não foi implementado qualquer algoritmo de cache para os acessos aos arquivos de *nonces*, a utilização de *buffered I/O* para este fim implica em ganhos de desempenho. Contudo, os processos que realizam operações de escrita do tipo *buffered I/O* sofrem influência do algoritmo *writeback throttling* (Corbet, 2010). A ideia básica desse algoritmo é que processos que sujam muitas páginas da *page cache* devem ser pausados por um período para não comprometer outros processos. Cada processo tem um limite de páginas que podem ser alteradas até que comece a ser pausado. A cada página escrita na *page cache*, o processo chama a função denominada `balance_dirty_pages` que verifica se o limite de páginas foi ultrapassado e calcula quanto tempo deve durar a pausa.

Como demonstrado na Seção 4.4, os processos que chamam as funções de escrita de *nonces* não são processos de usuário, mas sim *threads* de *kernel* que supostamente não sujam

páginas da *page cache* e consequentemente não deveriam sofrer com o *writeback throttling*. Dessa forma, em um cenário com muitas escritas assíncronas ou com uma carga de trabalho muito grande, as *threads* encarregadas de escrever os blocos de *nonces* podem ser pausadas, possivelmente causando *deadlocks*. Para solucionar este problema, duas abordagens podem aplicadas: realizar o *bypass* da pausa para *threads* que escrevem *nonces* ou realizar a escrita de *nonces* utilizando *direct I/O*.

Tendo em vista que as escritas de *nonces* são resultado de escritas de blocos de dados, o *throttling* dos processos que estão sujando páginas de dados influencia diretamente a quantidade de *nonces* escritos, dispensando a necessidade de pausar as *threads* de *kernel*. Para implementar esta abordagem, um *hook* na função `balance_dirty_pages` verifica a qual `inode` pertence a página que foi suja e, caso o `inode` em questão não seja um arquivo de dados, a *thread* não dorme.

Escritas de dados utilizando *direct I/O* não sujam páginas da *page cache*, consequentemente, não executam a função `balance_dirty_pages`. Para que fosse possível realizar a escrita de *nonces* utilizando *direct I/O* e continuar utilizando a mesma interface `kernel_write`, foi adicionado um *hook* na função em questão para preparar os parâmetros requeridos por esse tipo de operação.

Para que seja possível realizar a leitura dos arquivos de *nonces* utilizando *buffered I/O* e a escrita utilizando *direct I/O*, foram utilizados dois descritores de arquivos, já que o tipo da operação é definido no momento da abertura do arquivo. Os membros `iv_filp` e `iv_dio_filp` da estrutura `james_inodeinfo` são responsáveis por manter a referência para as estruturas que descrevem o *nonce file* aberto e são protegidos pelo semáforo `ivfile_lock`.

A coerência entre leituras que utilizam *buffered I/O* e escritas que utilizam *direct I/O* é tratada através de invalidações das páginas na *page cache*, o que diminui o desempenho em casos de cargas de trabalho com ambas as operações. Por isso, juntamente com a falta da implementação de um algoritmo de cache próprio para manter os *nonces* em memória e a necessidade de manter uma instância a mais de arquivo aberto para cada *nonce file*, utilizar *direct I/O* não apresentou bons resultados, tornando a abordagem do *bypass* da função de pausa mais conveniente.

A geração de *nonces* utilizados nas operações de escrita é realizada através de operações atômicas na estrutura denominada *Global Counter*, herdada do EncFS++. Quando requisitado, o *nonce* retornado é o valor atual do *Global Counter*: um número de 128 bits alinhado em 256. Cada requisição incrementa o *Global Couner em 256*. Estes 8 bits são reservados para que a Crypto API possa utilizá-los como contadores internos para aplicar o modo de operação CTR (que lida com blocos de 16 bytes) em blocos de 4096 bytes. Essa abordagem permite também que os bits de alta ordem sejam utilizados para armazenar um IV randômico sujerido para a utilização do modo de operação CTR. O esquema em questão é apresentado na Figura 4.6.

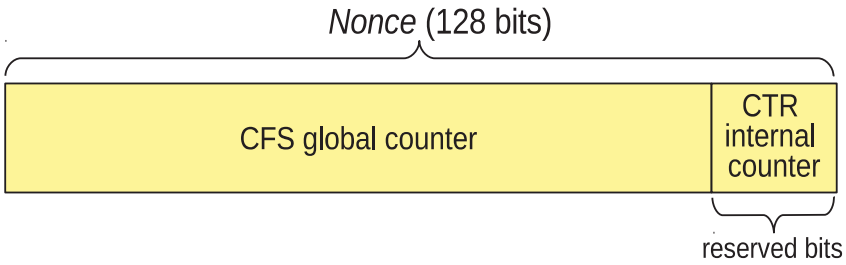



Figura 4.6: Estrutura do Global Counter (Eduardo et al., 2019).

## 4.4 FLUXOS DE EXECUÇÃO

Tendo em vista a utilização de *hooks* para implementar o SAC proposto, os fluxos de execução do EXT4 serão desviados para executar as operações criptográficas necessárias e retornarão ao fluxo normal. Algumas características dos fluxos de operações de leitura e escrita precisam ser consideradas: operações do tipo *buffered I/O* não se resumem somente ao tratamento das requisições do espaço de usuário, visto que as operações de E/S são realizadas de forma assíncrona, demandando o acompanhamento dessas operações por diferentes contextos; operações do tipo *direct I/O* possuem fluxos de execução diferentes que operações *buffered I/O*, e por isso, devem ser tratadas de forma diferente mas mantendo a consistência dos metadados entre ambos os tipos de operação. Além da leitura e escrita de dados, operações como abertura, fechamento, criação e remoção de arquivos também devem sofrer alterações com o objetivo de realizar um gerenciamento de metadados consistente.

### 4.4.1 Escrita de dados

A aplicação pode gerar operações de escrita do tipo *buffered I/O* ou do tipo *direct I/O*. Escritas do tipo *buffered I/O* atingem somente a *page cache*, mas podem gerar operações de E/S indiretamente, inclusive operações de leitura. Associado a cada bloco mapeado na *page cache* existe um bit denominado *dirty*, o qual indica se houve alteração nos dados desse bloco em memória. Essa alteração precisa, em algum momento, ser persistida no dispositivo de armazenamento. Essa persistência é realizada por *threads* de kernel denominadas *flusher threads* de forma assíncrona: em situações de pressão de memória, o *kernel* pode precisar retirar páginas da *page cache* para liberar espaço para aplicações; o bloco pode estar mapeado por muito tempo e precisa ser persistido para evitar perda de informações em casos de falhas. Se a escrita atingir um bloco que já contém dados mas que não está mapeado na *page cache*, o *kernel* realizará uma operação de E/S para ler esse bloco do dispositivo de armazenamento para a *page cache* e a escrita será feita em memória.

Operações do tipo *direct I/O* não passam pela *page cache* e utilizam o próprio *buffer* do espaço de usuário como a fonte dos dados para a operação de E/S. Como não é o *kernel* que cria o *buffer* que será usado na operação de E/S e não define quais os limites do arquivo a operação afetará, a aplicação deve seguir alguns requisitos: o endereço de memória do *buffer*, o *offset* de início da operação e o tamanho da operação devem ser alinhados ao tamanho do setor do disco, normalmente 512 bytes.

Com o objetivo de esconder a maior parte das singularidades de cada um dos tipos de operações, foi decidido realizar as operações de encriptação de dados no momento da requisição da operação de E/S, já que ambas passam por esse ponto. Além disso, utilizar a interface entre camada de sistemas de arquivos e a camada de E/S de blocos disponibiliza vantagens de ambos os contextos: todos os parâmetros passados para a camada de E/S de blocos estão prontos, contendo informações importantes que facilitam o acesso a endereços físicos e o acompanhamento da finalização da operação em questão; como este ponto ainda está na camada de sistema de arquivos, todas as informações referentes a ela estão disponíveis; tanto operações de *buffered I/O* quanto operações de *direct I/O* passam por esse ponto, possibilitando um tratamento similar. A Figura 4.7 apresenta a arquitetura do processo de escrita do SAC proposto.

Operações de escrita vindas do usuário que alteram somente a *page cache* não executam qualquer operação criptográfica. Já quando o *kernel* decide mover páginas da *page cache* para o dispositivo de armazenamento ou quando a aplicação realiza operações *direct I/O*, o sistema de arquivo EXT4 submete operações de escrita para a camada de E/S de blocos e os dados são encriptados. As máscaras que serão utilizadas para encriptar os dados são criadas a partir do

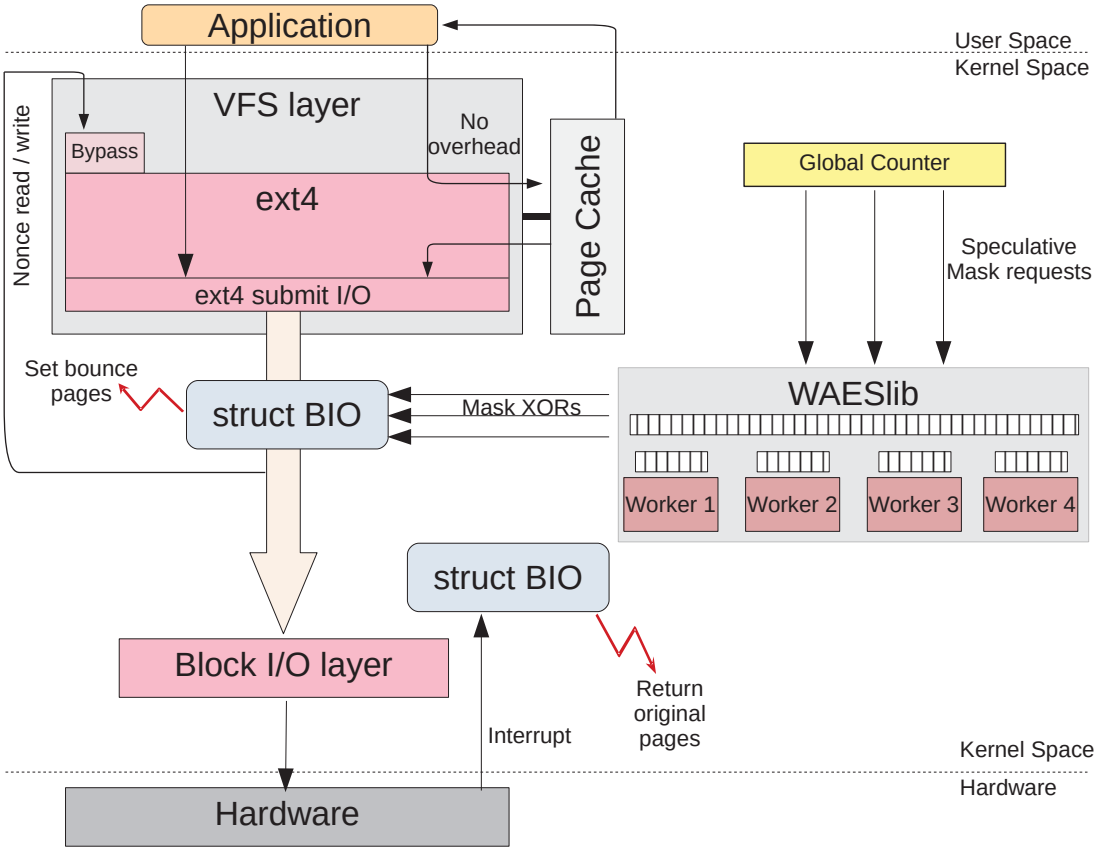



Figura 4.7: Arquitetura das operações de escrita.

*Global Counter* de forma especulativa, ou seja, assim que o sistema de arquivos é montado, máscaras já começam a ser criadas.

O *buffer* que armazena o resultado das operações criptográficas não pode ser no mesmo espaço de memória onde os dados limpos estão, pois isso implicaria na encriptação da *page cache* ou do *buffer* do usuário. Para resolver este problema, foi criado um *pool* de páginas físicas denominadas *bounce pages*, as quais são utilizadas para armazenar os resultados das operações criptográficas. A submissão das operações de E/S é feita através da estrutura `bio`, que além de outras informações, contém o endereço físico das páginas que contém os dados a serem escritos. Assim que a estrutura `bio` fica ponta para ser enviada para a camada inferior, o *hook* que realizada a operação criptográfica é chamado. Essa função busca páginas do *pool* de *bounce pages* e escreve o resultado da encriptação nessas páginas. Após isso, as referências contidas na estrutura `bio` são alteradas para referenciar as *bounce pages*, fazendo com que os dados escritos no dispositivo de armazenamento sejam os dados encriptados.

Ao final da operação de escrita, o *kernel* utiliza as páginas referenciadas pela estrutura `bio` para fazer operações de gerenciamento, como por exemplo apagar o bit *dirty* caso a página pertença à *page cache*. Dessa forma, antes que o *kernel* inicie qualquer operação, o *bio* deve voltar a apontar para as páginas originais. Por isso, quando o dispositivo de armazenamento dispara a interrupção de finalização da operação, um *hook* realiza a troca das referências e retorna a *bounce page* alocada para o *pool*. Cada *bounce page* mantém a referência para a página original que contém os dados decriptados no membro `private` da estrutura `page` (que descreve uma página física), permitindo a recuperação após a finalização da operação.

Como mencionado na Seção 4.3, para cada inode, é mantido um bloco de *nonces* em memória com o objetivo de diminuir a quantidade de acessos ao dispositivo de armazenamento.

Juntamente com o bloco de dados escrito, existe o *nonce* que foi utilizado na criação da máscara que foi aplicada sobre ele. Esse *nonce* deve ser armazenado para que o dado possa ser decriptado nas próximas operações de leitura. Caso o novo *nonce* pertença ao bloco de *nonces* em memória, a sua escrita será resumida a uma cópia de memória e o membro `flags` da estrutura `ivcache` indicará que o bloco está sujo. Caso o *nonce* em questão não pertença ao bloco em memória, ocorrerá a escrita do bloco mapeado (caso exista um bloco em memória e ele esteja sujo) e o bloco correto será trazido para a memória (caso ele já exista). Vale ressaltar que essa operação de busca do bloco de *nonce* gera uma operação de E/S, sendo relativamente custosa. Por isso, essas operação foi implementada de forma assíncrona, mantendo o semáforo leitores/escritores `ivcache_lock` travado para escrita.

Para que seja possível ler e escrever *nonces* no *nonce file*, este precisa estar aberto. Mantê-lo aberto enquanto o arquivo de dados está aberto é uma solução, contudo, como as operações de E/S que modificam a *page cache* acontecem de forma assíncrona, é possível que algumas delas aconteçam após o arquivo de dados ser fechado. De maneira análoga, os *nonces* também são escritos de forma assíncrona e podem gerar a mesma situação. Por isso, é necessário manter o controle da quantidade de blocos e de *nonces* a serem escritos (que podem ter valores diferentes caso operações de *buffered I/O* e *direct I/O* sejam aplicadas sobre o mesmo inode), para que, quando o último bloco e o último *nonce* forem escritos e não existir mais referências ao arquivo de dados, o *nonce file* seja fechado. O membro `blkwrite` da estrutura `james_inodeinfo` é utilizado para contabilizar a quantidade de blocos a serem escritos, o membro `ivcount` contabiliza a quantidade de *nonces* a serem escritos e o membro `refcount` mantém o número de referências ao arquivo de dados.

O fluxo das operações do tipo *direct I/O*, tanto para leitura quanto para escrita, é inicialmente tratado pelo VFS. Por isso, para que fosse possível chamar o *hook* fora do EXT4, o membro `james_on_dio_ptr` da estrutura `james_inodeinfo` é inicializado no momento de abertura do arquivo e utilizado para acessar a função desejada.

### 4.4.2 Leitura de dados

Uma das operações mais importantes do SAC proposto é a requisição de criação das máscaras criptográficas para as operações de leitura. O SAC deve submeter essas requisições de forma que seja muito provável que as máscaras estejam prontas quando os dados também estiverem prontos. É possível disparar a criação das máscaras criptográficas no nível de VFS, ou seja, assim que a requisição entra no espaço de *kernel*, contudo, essa abordagem não permite diferenciar quando uma requisição somente altera a *page cache* (onde os dados já estão decriptados) ou gera uma operação de E/S (onde a operação de decriptação é necessária). Levando isso em consideração, o ponto ideal para submeter requisições de criação de máscaras é a parte mais inferior da camada de sistemas de arquivos, quando ocorre a submissão das operações de E/S. Como este é o mesmo ponto utilizado para as operações de escrita, as vantagens citadas na Seção 4.4.1 também existem para as funções de leitura. Neste momento, os algoritmos de cache já decidiram que a operação de E/S é necessária, então não existe possibilidade de se gastar tempo de CPU criando máscaras desnecessárias. Como demonstrado na Seção 4.2, disparar criação de máscaras no ponto em questão ainda provê espaço suficiente para criação de máscaras. A Figura 4.8 apresenta a arquitetura das operações de leitura para o SAC proposto.

Diferentemente das operações de escrita, a operação de leitura precisa armazenar diversos dados referentes à operação de E/S em questão. Para isso, foi adicionado um membro na estrutura `bio` para referenciar a estrutura denominada `james_bioinfo`, apresentada no Código 4.3. Os membros dessa estrutura são descritos nos momentos oportunos.

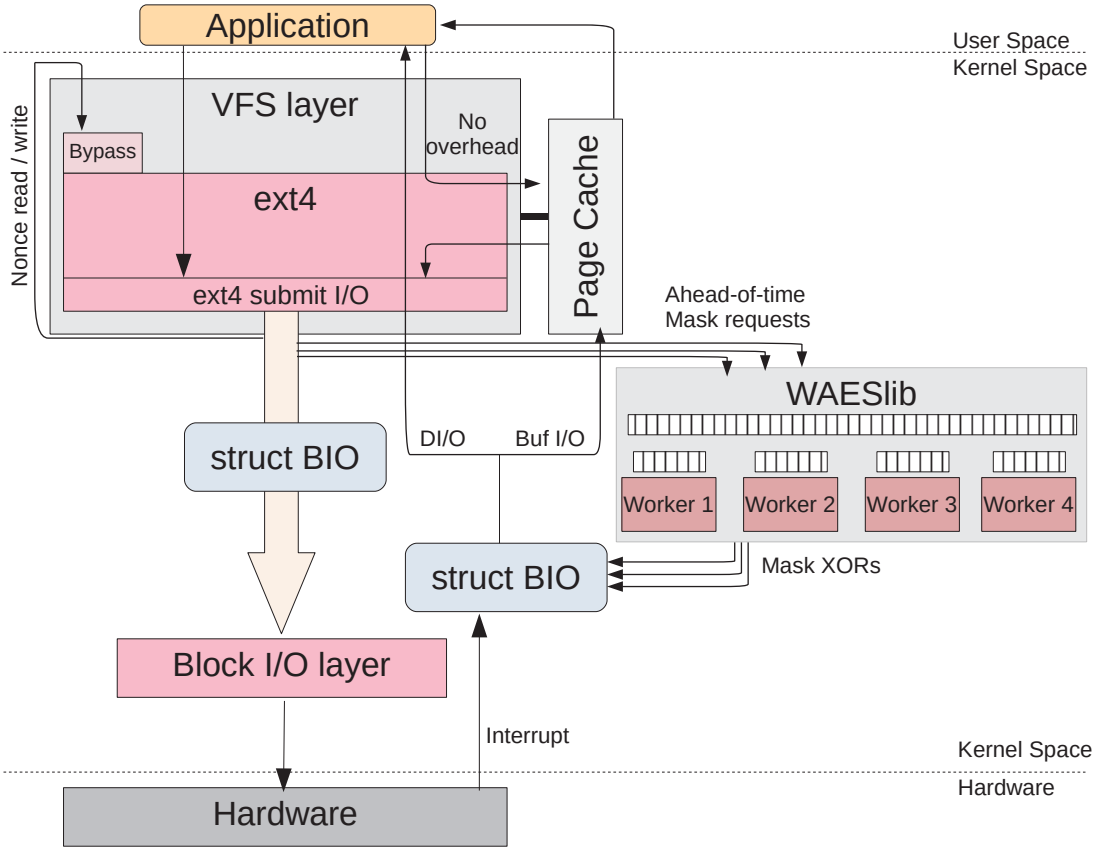



Figura 4.8: Arquitetura das operações de leitura.

Listing 4.3: Estrutura james_bioinfo.

```
1  struct james_bioinfo
2  {
3    struct bio *bio;
4    struct work_struct work;
5    uint32_t ctx_pool_num;
6    __uint128_t *ivs;
7    uint32_t nr_blocks;
8    uint32_t first_block;
9    bio_end_io_t *endio;
10   struct bvec_iter iter;
11   bool is_dio;
12   atomic_t mask_count;
13   atomic_t refcount;
14 #ifdef EXT4JAMES_BENCHMARK_RIO_ON
15   void *rio_chrono;
16 #endif
17 #ifdef EXT4JAMES_BENCHMARK_RPOSTIO_ON
18   void *rpostio_chrono;
19 #endif
20 };
```

Assim como para as operações de escrita, leituras vindas do usuário que atingem somente dados presentes na *page cache* não executam qualquer operação criptográfica. Para operações do tipo *direct I/O* e para leitura de blocos que vão popular a *page cache*, o sistema de arquivos cria uma estrutura `bio` e a submete para a camada de E/S de blocos. Logo antes dessa submissão, um *hook* é chamado para executar duas operações: ler os *nonces* dos blocos

envolvidos na operação de leitura (caso os *nonces* não estejam mapeados no bloco de *nonces* em memória) e referenciá-los através do membro *ivs*; disparar a criação das máscaras que serão usadas para decriptar os blocos de dados lidos. Os *nonces* a serem lidos dependem dos blocos de de dados envolvidos na operação. Esses blocos são referenciados pelos membros `nr_blocks` e `first_block`, preenchidos com base nos parâmetros das funções que os *hooks* foram inseridos: `ext4_mpage_readpages` para leituras de blocos para a *page cache* e `iomap_dio_bio_iter` para operações de *direct I/O*.

Não existe necessidade da utilização de *bounce pages* nas operações de leitura pois os *buffers* que receberão os dados encriptados lidos do dispositivo de armazenamento são os mesmos que esperam os dados decriptados, possibilitando a sobrescrita dos dados encriptados.

Durante o fluxo de execução da operação dentro da camada de E/S de blocos, uma estrutura `bio` pode gerar mais uma operação de E/S. Isso significa que as estruturas só podem ser desalocadas ao final da última operação, quando o membro `refcount` da estrutura `james_inodeinfo` atingir o valor 0.

Ao final de cada operação de E/S, o dispositivo de armazenamento dispara uma interrupção. Nesse momento, as máscaras de encriptação provavelmente estarão computadas e prontas para serem aplicadas. Visto que o contexto de interrupção não é ideal para realizar qualquer tipo de trabalho, um *hook* dispara a aplicação das máscaras através de *workqueues*, um mecanismo presente no *kernel* Linux que permite enfileirar trabalhos (funções) que serão executadas por *threads* de *kernel* criadas previamente. Essa abordagem permite, também, que as *threads* durmam, caso de alguma máscara ainda não estar pronta para ser aplicada. O membro `work` da estrutura `james_bioinfo` é utilizado para passar toda a estrutura `james_inodeinfo` como parâmetro para a *thread* que executará a aplicação das máscaras, enquanto o membro `mask_count` mantém o números de máscaras prontas para o `bio` em questão.

Após aplicação de todas as máscaras da operação em questão, o *hook* chama a função referenciada pelo membro `endio` que aponta para função que é originalmente chamada pela interrupção para indicar às camadas superiores que o dado já esta pronto, fazendo com que este seja repassado para o usuário.

### 4.4.3 Demais operações

As operações de leitura e escrita de blocos são as operações mais importantes dentro do SAC proposto, contudo, para que os metadados se mantenham coerentes com o resto do sistema de arquivos, outras operações precisam disparar *hooks* que realizam operações de gerenciamento.

No momento que o SAC é montado, o *Global Counter*, armazenado nos *extended attributes* do *Global File*, é lido e armazenado em memória. Como as estruturas `james_inodeinfo`, `james_bioinfo` e `james_ivcache` são comumente alocadas, foi utilizado o mecanismo *Slab Cache Allocator*, que cria uma cache de estruturas e provê melhor desempenho nas alocações. Além dessas estruturas, o *hook* na operação de montagem do SAC também cria uma *Slab Cache* da estrutura denominada `james_ivcache_async`, demonstrada no código 4.4, que é utilizada para agrupar os parâmetros necessários para a escrita assíncrona de um *nonce*. Ademais, neste mesmo *hook*, o *pool* de *bounce pages* utilizado nas operações de escritas é criado, a WAESlib é inicializada juntamente com a criação do *pool* de contextos de encriptação (descrito na Seção 4.5). Como citado anteriormente, operações de decriptação de blocos e escrita de *nonces* podem ser realizadas de forma assíncrona. Para isso, são utilizadas *workqueues*, que são criadas no momento da montagem do SAC. Cada *workqueue* possui a quantidade de *threads* trabalhadoras igual ao número de processadores disponíveis na máquina e possuem alta prioridade, visto que são trabalhos rápidos ou operações de E/S. Quando o SAC é desmontado, o *Global Counter*

é persistido no dispositivo de armazenamento, a WAESlib é finalizada, as *Slab Caches* são destruídas, assim como o *pool* de *bounce pages* e as *workqueues*.

Listing 4.4: Estrutura james_ivcache_async.

```
1 struct james_ivcache_async
2 {
3       struct james_inodeinfo *inodeinfo;
4       struct work_struct work;
5       __uint128_t iv;
6       uint32_t iv_pos;
7 };
```

Quando um arquivo de dados é criado, um *hook* é disparado com o objetivo de reservar um bloco do *Global File* e armazenar o endereço retornado nos *extended attributes*. O *nonce file* dedicado para esse arquivo só é criado quando necessário, ou seja, em operações de escritas que façam o arquivo crescer além de 1MB. Quando o arquivo é deletado, o bloco reservado do *Global File* é liberado e o *nonce file* é deletado, caso exista. Além disso, o membro *unlink_called* presente na estrutura `james_inodeinfo` é definido com o objetivo de invalidar qualquer operação assíncrona que esteja pendente.

Quando um arquivo de dados é aberto pela primeira vez, uma estrutura `james_inodeinfo` é alocada e inicializada. Nas próximas aberturas, somente o contador de referências é incrementado. No fechamento dos arquivos de dados, somente o contador de referências é decrementado. A estrutura `james_inodeinfo` não é desalocada quando a última referência ao arquivo é perdida, da mesma forma que a estrutura `inode`. Isso acontece pois é provável que este arquivo seja novamente referenciado futuramente. Quando o EXT4 decide retirar um `inode` da memória, um *hook* é disparado com o objetivo de desalocar a estrutura *james_inodeinfo* e a estrutura `james_ivcache` que mantém o bloco de *nonces* em memória.

## 4.5 ADAPTAÇÃO DA WAESLIB E GERENCIAMENTO DOS CONTEXTOS CRIPTOGRÁFICOS

A WAESlib é responsável por prover criptografia especulativa e *ahead-of-time* de forma trivial através da sua simples interface, majoritariamente composta por funções de submissão de criação de máscaras e utilização das mesmas. Usuários da WAESlib alocam um contexto criptográfico, definem o *nonce*, a chave e o *buffer* onde a máscara criada será armazenada, então, submetem a requisição de criação da máscara para o contexto em questão. Para utilizar a máscara criada em operações de encriptação ou decriptação, o usuário informa o *buffer* que contém o dado a ser submetido à operação criptográfica e o contexto que contém a máscara para a função da WAESlib que realiza a operação de XOR. Visto que a WAESlib foi originalmente implementada para executar em espaço de usuário, não é possível utilizá-la nesta proposta sem uma completa reimplementação.

O primeiro ponto a se considerar é que a implementação original da WAESlib utiliza a GPU para criar máscaras criptográficas através da biblioteca CUDA (Nickolls et al., 2008). Por esta ser uma biblioteca de usuário, o *kernel* não consegue acessar diretamente as interfaces CUDA, o que gera necessidade da existência de um *daemon* em espaço de usuário para realizar as requisições ou utilizar alguma interface como o KGPU (Sun et al., 2012). Contudo, essas soluções implicam no aumento na latência das operações. Caso o código do *driver* da Nvidia fosse aberto, seria possível realizar requisições diretas a ele, mas esse não era o caso durante o projeto do sistema proposto. Por isso, esta proposta utiliza as tecnologias presentes nos

processadores *multicore* para criação das máscaras, objetivando baixa latência da comunicação entre as *threads* trabalhadoras da WAESlib e as *threads* clientes.

O contexto criptográfico, apresentado no Código 4.5, é a estrutura central da WAESlib. Ela principalmente agrega a referência à chave criptográfica, *buffer* que contém a máscara, e *nonce* (nos membros `key_number` e `key_index`, `mask` e `iv`, respectivamente), mas também contém diversas estruturas que são utilizadas internamente pela WAESlib.

Listing 4.5: Estrutura waes_context.

```
struct waes_context
{      uint32_t ctx_number;
       uint32_t key_number;
       uint32_t key_index;
       __uint128_t iv;
       uint8_t *mask;
       struct crypto_skcipher *tfm;
       struct skcipher_request *req;
       struct scatterlist *sg;
       uint8_t priority;
       uint8_t thread_id;
       uint8_t status;
       struct mutex mutex_ready;
       atomic_t *mask_count;
       bool ctx_reserved;
};
```

Para realizar a criação das máscaras criptográficas, foi utilizada a Crypto API (Mueller e Vasut, 2023), uma biblioteca em espaço de *kernel* que provê as seguintes vantagens: utilização da tecnologia AES-NI (Gueron, 2010), quando disponível, de forma transparente e otimizada; utilização de chaves de tamanhos variados, permitindo a fácil integração com diversos mecanismos de gerenciamento de chaves; interfaces para lidar com endereços físicos, possibilitando o tratamento de *buffers* de espaço de usuário, vindos de operações *direct I/O*, e *buffers* de espaço de *kernel*, vindos de operações *buffered I/O*, de forma similar. O membro `tfm` da estrutura `waes_context` define a cifra utilizada pela Crypto API, nesse caso o algoritmo AES com instruções AES-NI no modo de operação CTR. O membro `req` referencia a estrutura utilizada para requisitar a execução da cifra em questão. O membro `sg` é utilizado pada descrever o dado envolvido na operação criptográfica, visto que a Crypto API utiliza *scatterlists* como *buffers*.

A Crypto API não consegue efetuar somente o passo de criação de máscaras do modo de operação CTR, pois a cifra completa deve ser executada. Com o objetivo de gerar somente a máscara, o dado de entrada passado para operação sempre é um *buffer* com bytes 0, pois dessa forma o XOR não gera modificações, fazendo com que o resultado final seja a máscara. Futuramente, é possível inserir uma transformação na Crypto API que realize somente a criação de máscaras utilizando instruções AES-NI, com o objetivo de diminuir a latência de criação das máscaras.

A operação de XOR, executada entre o texto claro e a máscara criptográfica, pode ser aplicada de diferentes formas, como por exemplo através de instruções vetoriais. Para isso, foi utilizado o módulo `xor` do *kernel* Linux, que identifica o método mais rápido para realizar as operações e as utiliza quando a interface `xor_blocks` é chamada. No caso da máquina utilizada neste trabalho, as operações de XOR foram executadas através de instruções AVX.

Visto que as máscaras criptográficas são criadas de forma assíncrona, a WAESlib precisa manter um conjunto de *threads* trabalhadoras para realizar este trabalho. Quando ocorre uma requisição de criação de máscara, o contexto passado por parâmetro é enfileirado. Com o objetivo de implementar o conceito de prioridade entre os contextos submetidos, a fila de contextos

foi implementada através de uma *heap* binária, tornando-a uma fila de prioridade. As *threads* trabalhadoras, por sua vez, executam um laço infinito decrementando um semáforo que age como contador dos contextos na fila de prioridade. Dessa forma, quando o contador do semáforo atinge zero, a *thread* trabalhadora que tenta buscar um contexto na fila é bloqueada até que novos contextos sejam enfileirados.

Como apresentado no Capítulo 5, em situações em que o dispositivo possui uma latência muito baixa, como a memória RAM, e o número de requisições é muito alto, a utilização de uma única fila de contextos pode se tornar um gargalo pois todas as *threads* trabalhadoras vão acessar ao mesmo tempo uma única fila. Por isso, foi implementado também uma fila para cada *thread* trabalhadora, de modo que os contextos são enfileirados de forma cíclica entre eles.

Um contexto criptográfico pode estar em diferentes estados durante a execução do SAC, de forma que o membro `status` da estrutura `waes_context` é utilizado para identificar o estado atual. Quando o contexto é criado pela WAESlib, ele recebe o estado `WAES_CTX_STATUS_IDLE`. No momento que ocorre uma requisição de criação de máscaras para o contexto em questão, o seu estado é alterado para `WAES_CTX_STATUS_INQUEUE`. Quando uma *thread* trabalhadora busca esse contexto e inicia o preparo da máscara, o estado é alterado para `WAES_CTX_STATUS_INPROCESS`. Ao final da criação, o estado do contexto volta a ser `WAES_CTX_STATUS_IDLE`. A WAESlib permite que uma máscara no estado `WAES_CTX_STATUS_INQUEUE` receba uma nova requisição ou somente mude a sua prioridade, causando um rebalanceamento da *heap* binária.

As funções que compõem a interface da WAESlib foram mantidas, mudando somente os nomes: `waes_init` inicializa os contextos criptográficos, inicializa as estruturas da Crypto API e dispara as *threads* trabalhadoras, enquando a função `waes_exit` desaloca todas as estruturas e finaliza as *threads* trabalhadoras. A função `waes_request_mask` é utilizada para submeter uma requisição de criação de máscara, enquanto a função `waes_encrypt_decrypt` é utilizada para aplicar a máscara criada, caso ela esteja pronta. Chaves criptográficas são registradas através da função `waes_set_key`.

É possível que o usuário da WAESlib chame a função `waes_encrypt_decrypt` antes que a máscara esteja pronta. Com o objetivo de evitar a aplicação de uma máscara que ainda não está pronta, a função `waes_encrypt_decrypt` tenta travar o *mutex* `mutex_ready` presente na estrutura `waes_context`. Este, por sua vez, é travado no momento da requisição da máscara e destravado quando a máscara está pronta. Dessa maneira, o processo que tenta aplicar uma máscara que ainda não está pronta dorme até que esta fique pronta. A WAESlib ainda provê uma segunda forma de verificação das máscaras prontas. O membro `mask_count` é um ponteiro que aponta para um contador definido pelo usuário da biblioteca. A chamada da função `waes_request_mask` recebe como um dos parâmetros o endereço desse contador que será incrementado após cada criação de máscara, possibilitando o processo usuário saber quantas máscaras requisitadas foram criadas. Isso pode evitar que o processo durma em contextos como interrupções, que são atômicos, além de permitir um gerenciamento mais detalhado dos contextos.

A WAESlib provê os contextos criptográficos de forma individual, com o propósito de que a aplicação usuária seja livre para gerenciá-los da forma mais conveniente. O EncFS++ propôs a utilização de dois *pools* de contextos criptográficos para cada arquivo aberto (um para leituras sequenciais e um para leituras randômicas) e um *pool* global para operações de escrita, utilizado por todos os arquivos do SAC. Os *pools* eram encarados como janelas deslizantes, como apresentado na Seção 4.1.

Tendo em vista que a arquitetura proposta não lida com requisições de leitura e escrita no nível de arquivo, mas sim em nível de requisições de disco, não faz sentido manter *pools* de

contextos para cada instância de arquivo aberto. Por isso, para as operações de leitura, foi definido um *pool* para cada estrutura `bio` criada. O cenário ideal seria criar um *pool* com a quantidade de contextos igual ao número de páginas da requisição em questão, possibilitando a criação do máximo de máscaras durante a operação de E/S e dispensando a necessidade da abordagem da janela deslizante. Contudo, um conjunto de processos realizando grandes requisições de leitura poderiam reservar um grande número de contextos, desbalanceando a utilização com os demais. Portanto, o SAC define um tamanho máximo para os *pools* de leitura e, caso a requisição ultrapasse essa quantidade de blocos e não seja quebrada em requisições menores, o *pool* é tratado como uma janela deslizante.

Leituras sequenciais e randômicas eram tratadas por diferentes *pools* pois as máscaras eram criadas especulativamente. Uma leitura fora de sequência causava a invalidação das máscaras já criadas e gerava novas requisições criação, por isso, as leituras randômicas eram tratadas por um *pool* menor que não influenciavam as máscaras criadas para as operações de leitura. Na proposta atual, as máscaras são criadas de forma *ahead-of-time*, permitindo o tratamento de padrões sequenciais e randômicos de acesso da mesma forma, diminuindo a quantidade de contextos reservados e evitando por completo a invalidação de máscaras criadas.

Já para operações de escrita, o esquema proposto pelo EncFS++ é aplicável na proposta atual. Um *pool* de contextos global que pode ser utilizado por todos os arquivos no SAC é ideal caso todos os arquivos do sistema sejam encriptados com a mesma chave. Essa proposta ainda habilita a utilização de criptografia por arquivos, por diretório e por usuário, dado seu contexto de execução. A quantidade de *pools* de escrita depende da quantidade de chaves criptográficas utilizadas em um determinado momento.

Nas operações de leitura, as máscaras necessárias em uma operação de E/S são requisitadas dentro de um laço. Nas operações de escrita, as requisições de máscaras ocorrem de forma intercalada com a aplicação das mesmas. Esse tempo entre as requisições de máscaras nas operações de escrita faz com que as *threads* trabalhadoras durmam e acordem em um intervalo muito pequeno, de forma que o tempo gasto para trocar os estados de execução cria um *overhead* considerável. Para contornar esse problema, o membro denominado `reserved_ctx` é utilizado para reservar contextos que as máscaras serão aplicadas, e em seguida, as máscaras para estes contextos são requisitadas em um laço de uma só vez.

# 5 RESULTADOS

Com o objetivo de verificar a vantagem de aplicar a criptografia especulativa e *ahead-of-time* em um contexto de SAC *cross-stack*, o sistema de arquivos padrão do Linux, EXT4, foi modificado de acordo com a arquitetura apresentada nas seções anteriores, de forma que o novo SAC *proof-of-concept* EXT4james foi criado. Neste capítulo, o EXT4james é comparado com o fscrypt (Kernel Development Community, 2023f), um SAC implementado em espaço de *kernel* através de *hooks* dentro do EXT4, F2FS e UBIFS, que possibilita a utilização do algoritmo de criptografia AES nos modos XTS e CBC através da biblioteca Crypto API. O fscrypt foi escolhido para realizar as comparações pois ele executa em um contexto similar ao SAC proposto, possibilitando o enfoque das análises no *overhead* das operações criptográficas.

Foi analisado o desempenho tanto para operações de leitura quanto para operações de escrita, executadas sobre um dispositivo de armazenamento NVMe e um dispositivo de bloco baseado em RAM. Todos os testes descritos foram executados em um sistema com *kernel* Linux versão 6.1.10 em uma máquina equipada com um processador Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz, memória (RAM) 16GB 2933MHz DDR4, e NVMe ADATA SX6000LNP. Todos os testes foram executados 10 vezes durante 60 segundos, o que gera movimentações de mais de 1TB de dados para operações de leituras sequenciais, por exemplo. As operações foram executadas utilizando *direct I/O*, objetivando a geração de operações de E/S a cada requisição. A tecnologia *hyper-threading* foi desativada, com o objetivo de evidenciar a escalabilidade em núcleos físicos e facilitar a análise em ambientes de múltiplos clientes. O *Linux governor*, que controla a escala de frequência do processador, foi definido para o modo *performance*.

O fscrypt começou a prover suporte à *direct I/O* somente na versão 5.18 do *kernel* Linux, através da interface `blk-crypto`. Através dessa interface, a encriptação passa a ser executada no nível de E/S de blocos, como o dm-crypt. O `blk-crypto` utiliza *Inline Encrypt Hardware* (Kernel Development Community, 2023g) para realizar as operações criptográficas, contudo, esta funcionalidade não está disponível na máquina utilizada para os testes, de forma que a implementação *fallback* que utiliza Crypto API foi executada. Vale ressaltar também que o fscrypt impõe algumas condições restritivas para a utilização de *direct I/O* relacionadas aos alinhamentos. Enquanto a camada de sistemas de arquivos requer alinhamento ao tamanho do setor, o fscrypt requer alinhamento ao tamanho da página, fazendo com que algumas aplicações precisem alterar seus códigos.

Para permitir a análise mais detalhada das operações de leitura e escrita, foi utilizada a ferramenta `FIO` (Axboe, 2023) e suas cargas de trabalho de *micro-benchmark*: leituras e escritas sequenciais e randômicas. A ferramenta `pidstat` (Sebastien, 2023) foi utilizada para verificar a utilização dos núcleos da CPU pelas *threads* trabalhadoras e a ferramenta Filebench (Vasily, 2016) foi utilizada para realizar testes de *macro-benchmark* através de suas cargas de trabalho sintéticas que simulam aplicações reais: *fileserver* e *webserver*.

As Seções 5.1 e 5.2 apresentam a análise os resultados obtidos com a aplicação das cargas de trabalho de *micro-benchmark*. A Seção 5.3 apresenta os resultados obtidos com a aplicação das cargas de trabalho em um ambiente que não possui aceleração em hardware para as operações criptográficas. A Seção 5.4 apresenta as limitações de hardware identificadas durante os testes das seções anteriores e que restringem as possibilidades de testes nos próximos cenários. A Seção 5.5 apresenta os resultados obtidos para um cenário de múltiplos clientes e a Seção 5.6 apresenta os resultados obtidos através da aplicação das cargas de trabalho de *macro-bencharmk*.

## 5.1 OPERAÇÕES DE LEITURA

A Figura 5.1a apresenta a vazão de requisições de leitura sequenciais realizadas para um arquivo de 10G, armazenado no dispositivo NVMe citado anteriormente, gerenciado pelo sistema de arquivos EXT4 (sem criptografia, representando os valores ideais), EXT4james e fscrypt. No eixo Y esquerdo está representada vazão em MB/s enquanto o eixo Y direito apresenta a melhora de desempenho do EXT4james comparado com o fscrypt. A Figura 5.1b apresenta a latência das operações.

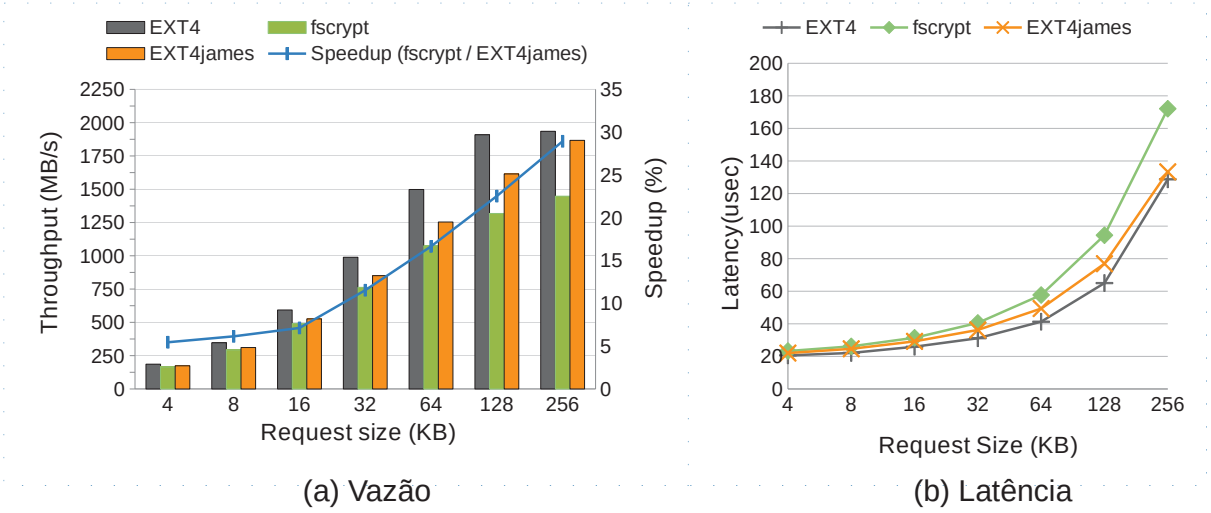



Figura 5.1: Leitura sequencial (dispositivo NVMe).

Os resultados mostram que o EXT4james melhorou a vazão em até 28% e diminuiu a latência em até 22% para esse cenário. Para facilitar o aprofundamento nesses resultados, a Tabela 5.1 apresenta outros valores para a carga de trabalho em questão. O tempo de E/S é contado a partir da submissão do `bio` para a camada de E/S de blocos até a a geração de interrupção do dispositivo de armazenamento que sinaliza o final da operação. Dado o tamanho da operação de E/S e seu tempo médio, é possível calcular quanto tempo existe para criar as máscaras criptográficas.

Tabela 5.1: Tempo de E/S para requisições de leitura sequenciais no EXT4james (dispositivo NVMe).

| Request size (KB) | Average I/O time (ns) | Average ns per mask | Average core load(%) | Average sleep time (ns) |
|---|---|---|---|---|
| 4 | 18269 | 18269 | 3.69 | 38389 |
| 8 | 20026 | 10013 | 6.39 | 22208 |
| 16 | 23579 | 5895 | 9.08 | 13562 |
| 32 | 28623 | 3578 | 14.40 | 8208 |
| 64 | 38378 | 2399 | 19.57 | 5138 |
| 128 | 58439 | 1826 | 25.56 | 3764 |
| 256 | 103526 | 1617 | 28.73 | 3201 |

Como apresentado na Seção 4.2, o tempo médio de criação de uma máscara é de 1192 ns. Mesmo quando as requisições são maiores, diminuindo do tempo de E/S por bloco, ainda existe tempo suficiente para criar todas as máscaras de forma *ahead-of-time*. Vale ressaltar que quanto maior o tamanho da requisição, melhor o desempenho do EXT4james quando comparado com o fscrypt. Isso se dá por duas principais questões: possibilidade de agregação de requisições de máscaras e maior aproveitamento do tempo de E/S.

Quando requisições de E/S grandes são executadas, mais máscaras são requisitadas à WAESlib de uma só vez. Isso faz com que as *threads* trabalhadoras possam trabalhar por mais

tempo sem dormir frequentemente. Essa análise também pode levar em consideração a utilização dos núcleos de processamento pelas *threads* trabalhadoras. É perceptível que a utilização média não ultrapassou 28%, ou seja, não existe uma sobrecarga de trabalho para requisições grandes. Esses resultados foram obtidos através da utilização de duas *threads* trabalhadoras buscando as requisições em uma única fila. Como não existe sobrecarga de trabalho, adicionar mais *threads* trabalhadoras faria com que cada *thread* dormisse mais vezes, diminuindo o desempenho.

Quanto maior a requisição, menor é o tempo de E/S por bloco. Isso faz com que o tempo das operações criptográficas se torne mais relevante, visto que o *overhead* de E/S diminui. Quanto mais essa proporção se acentua, maior será a porcentagem tempo de E/S aproveitado para criação de máscaras, cada vez mais escondendo a latência que impacta no desempenho geral do sistema.

Vale ressaltar também que, entre as requisições de 128KB e 256KB, a vazão do EXT4 não acompanha o crescimento dos outros tamanhos de requisições, fazendo com o que o EXT4james se aproxime bastante do seu desempenho. Isso se dá pois o limite da vazão especificado pelo fabricante do dispositivo NVMe foi atingido, limitando tanto o desempenho do EXT4 quanto o desempenho do EXT4james.

Para melhor entendimento do impacto da utilização do tempo de E/S para criação de máscaras e o limite imposto pela vazão do NVMe, a Figura 5.2 mostra a mesma carga de trabalho executada para os mesmos sistemas de arquivos, mas acima de um dispositivo de blocos baseado em RAM.

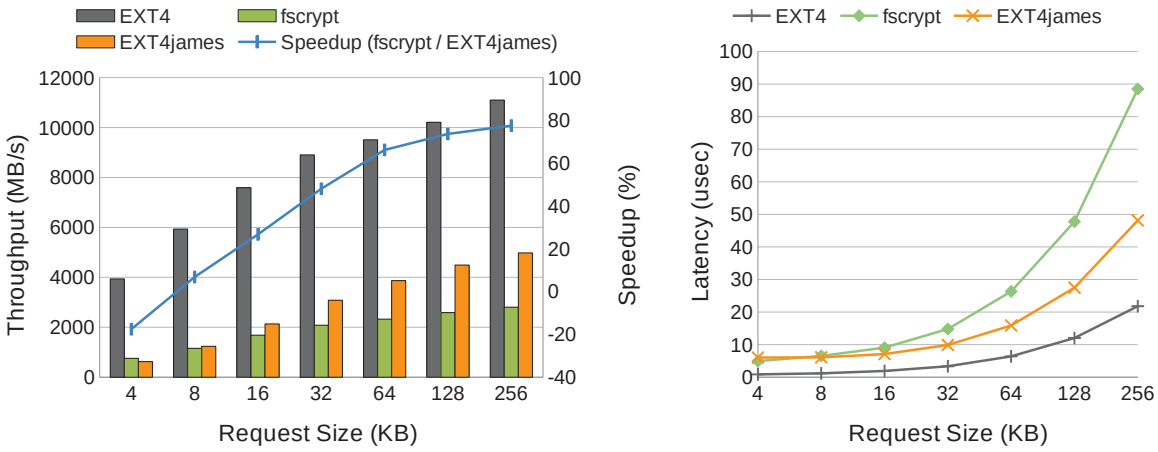



Figura 5.2: Leitura sequencial (dispositivo de blocos RAM).

Analisando esses resultados, é possível verificar que para um dispositivo de armazenamento de baixa latência, o EXT4james consegue melhorar a vazão em até 77% e diminuir a latência em até 45%. A maior agregação de máscaras e a utilização total do tempo de E/S faz com que o EXT4james seja mais eficiente quando comparado com o fscrypt.

Embora o EXT4james tenha melhorado o desempenho comparado com o fscrypt, o cenário de um dispositivo de blocos baseado em RAM possui algumas desvantagens. O aumento da diferença entre o EXT4james e o EXT4 pode ser atribuído a dois fatores: como mostrado na Tabela 5.2, o tempo de E/S é menor do que o necessário para criar as máscaras, o que faz o EXT4james tirar total proveito do tempo de E/S mas não ser capaz de esconder totalmente a latência das operações criptográficas, de forma que as *threads* clientes precisem esperar no momento de aplicação das máscaras; a utilização da memória RAM como um dispositivo de armazenamento faz com que a operação de E/S seja realizada pela própria CPU e não por outro

hardware, fazendo com que as *threads* trabalhadoras compitam pela CPU com as requisições de E/S, visto que estas não liberam o processador. Isso significa que, mesmo em um cenário adverso, onde não existe tempo ocioso durante a operação de E/S, o EXT4james consegue utilizar a criação de máscaras *ahead-of-time* tirando vantagem dos outros núcleos de processamento da CPU.

Tabela 5.2: Tempo de E/S para requisições de leitura sequenciais no EXT4james (dispositivo de blocos RAM).

| Request size (KB) | Average I/O time (ns) | Average ns per mask | Average core load (%) | Average sleep time (ns) |
|---|---|---|---|---|
| 4 | 411 | 411 | 18.07 | 15186 |
| 8 | 696 | 348 | 26.7 | 4834 |
| 16 | 1355 | 338.75 | 34.63 | 2947 |
| 32 | 2551 | 318.875 | 45.69 | 1712 |
| 64 | 4919 | 307.4375 | 58.56 | 1021 |
| 128 | 9762 | 305.0625 | 68.87 | 657 |
| 256 | 19755 | 308.671875 | 76.28 | 497 |

O cenário ideal para teste seria utilizar o SAC proposto em conjunto com dispositivos de armazenamento de baixa latência, pois eles liberam a CPU para realizar as operações de E/S e diminuem consideravelmente o impacto dessas operações , possibilitando o SAC proposto otimizar uma maior porcentagem da latência total. Infelizmente, esse hardware não está disponível.

Vale ressaltar também que, para requisições de 4KB, o EXT4james apresenta um desempenho 20% pior que o fscrypt. Isso ocorre porque o tempo de E/S é muito rápido, de forma que a troca de contextos entre as *threads* clientes e trabalhadoras, espera pela finalização da máscara e a troca de contexto para a *thread* da *workqueue* que realiza a operação do XOR se tornam muito grandes e acabam não compensando. Futuramente, é possível otimizar esse cenário através da execução da criptografia no modo CTR padrão, sem a utilização da WAESlib. Dessa forma, a própria *thread* cliente executaria a operação e não geraria trocas de contextos adicionais, além de apresentar um potencial de melhora da escalabilidade em cenários de múltiplos clientes.

A Figura 5.3 apresenta os resultados obtidos na execução da leitura randômica tanto para o dispositivo NVMe quanto para o dispositivo de bloco em RAM. Os resultados apresentaram um aumento na vazão de até 9% e 78% no NVMe e no dispositivo de bloco em RAM, respectivamente, e a diminuição da latência em até 8% e 43%, respectivamente. Os resultados obtidos para as requisições de 256KB no dispositivo NVMe não foram reportados pois, para esse tamanho em leituras randômicas, o EXT4 apresenta uma grande variação de desempenho (que reflete tanto no EXT4james quanto no fscrypt), impossibilitando analisar o *overhead* das operações criptográficas.

É perceptível que o ganho neste cenário para os dispositivos NVMe é muito menor. Isso se dá pelo fato de que as operações de E/S são muito longas, como mostra a Tabela 5.3, fazendo com que a parte otimizada não impacte tanto no desempenho geral. Para uma requisição de 4KB, por exemplo, o tempo de E/S é suficiente para criar mais de 100 máscaras. Já no cenário baseado em RAM, o resultado é similar ao sequencial, mostrando que o tratamento similar entre requisições sequenciais e randômicas apresentado na Seção 4.5 é eficiente.

## 5.2 OPERAÇÕES DE ESCRITA

A Figura 5.4 apresenta os resultados obtidos para a carga de trabalho de escritas sequenciais. Analisando os gráficos, pode-se verificar um aumento da vazão em até 11% e 81%, para o dispositivo NVMe e RAM, respectivamente, e a diminuição da latência em 9% e 45%, respectivamente.

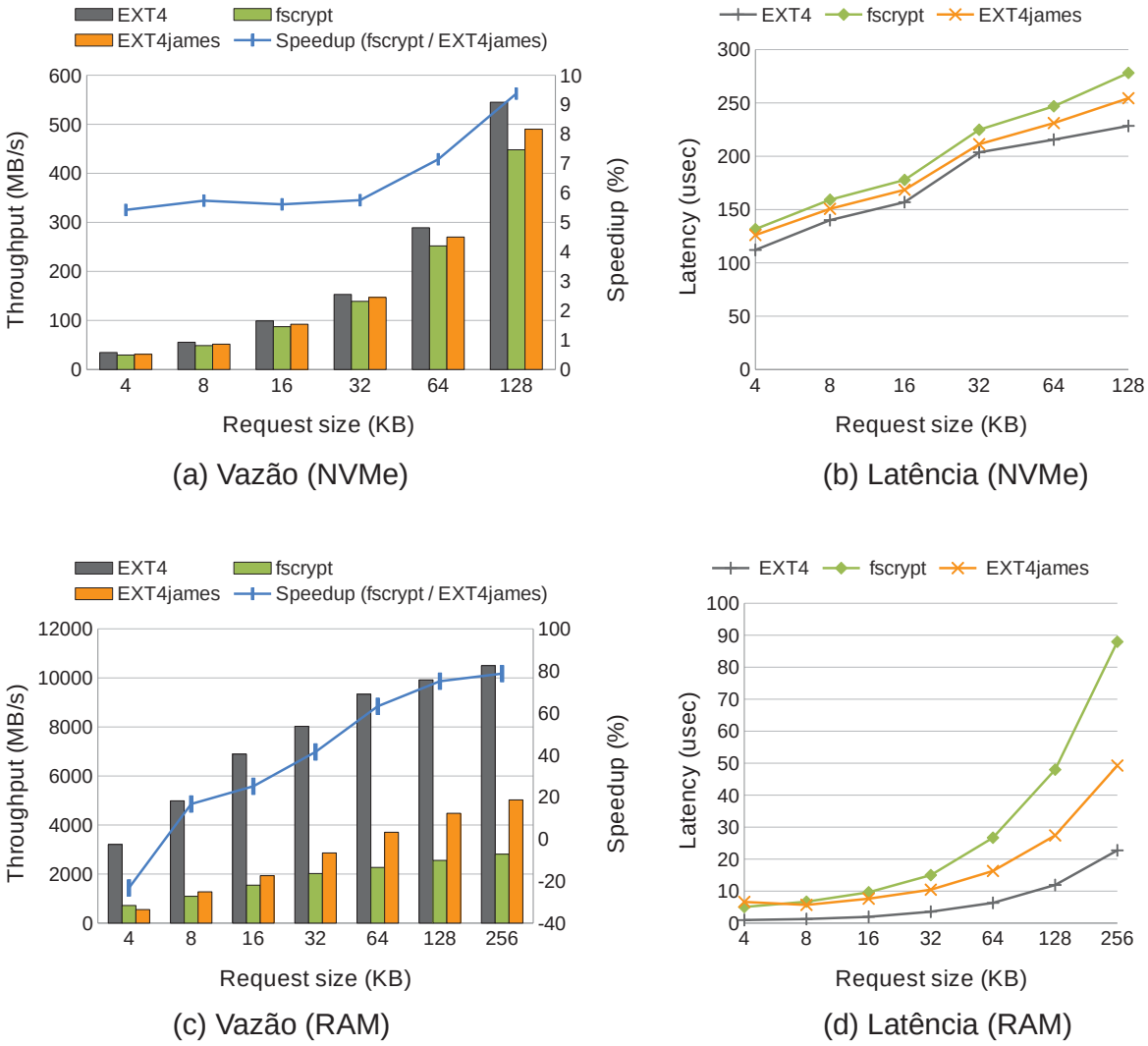



Figura 5.3: Leitura randômica.

Tabela 5.3: Tempo de E/S para requisições de leitura randômica no EXT4james (dispositivo NVMe).

| Request size (KB) | Average I/O time (ns) | Average ns per mask | Average core load (%) | Average sleep time (ns) |
|---|---|---|---|---|
| 4 | 121777 | 121777 | 0.67 | 122203 |
| 8 | 145598 | 72799 | 1.09 | 95501 |
| 16 | 162068 | 40517 | 1.89 | 60395 |
| 32 | 202648 | 25331 | 2.34 | 44265 |
| 64 | 218170 | 13636 | 4.15 | 24500 |
| 128 | 235138 | 7348 | 7.68 | 14791 |

Percebe-se que a melhora de desempenho da escrita sequencial em dispositivos NVMe é inferior à apresentada nas leituras sequenciais. Contudo, o cenário se inverte quando o dispositivo baseado em RAM é utilizado. Isso ocorre devido ao limite da vazão imposta pelo NVMe, fazendo com que o desempenho do fscrypt consiga se aproximar dos demais sistemas.

A melhora do desempenho da escrita sequencial, quando comparada com a leitura sequencial, se dá pela criação especulativa de máscaras utilizadas na escrita. Diferentemente das operações de leitura, máscaras podem ser criadas antes mesmo de existirem requisições, fazendo com que seja muito provável que exista uma máscara pronta para utilização quando necessário.

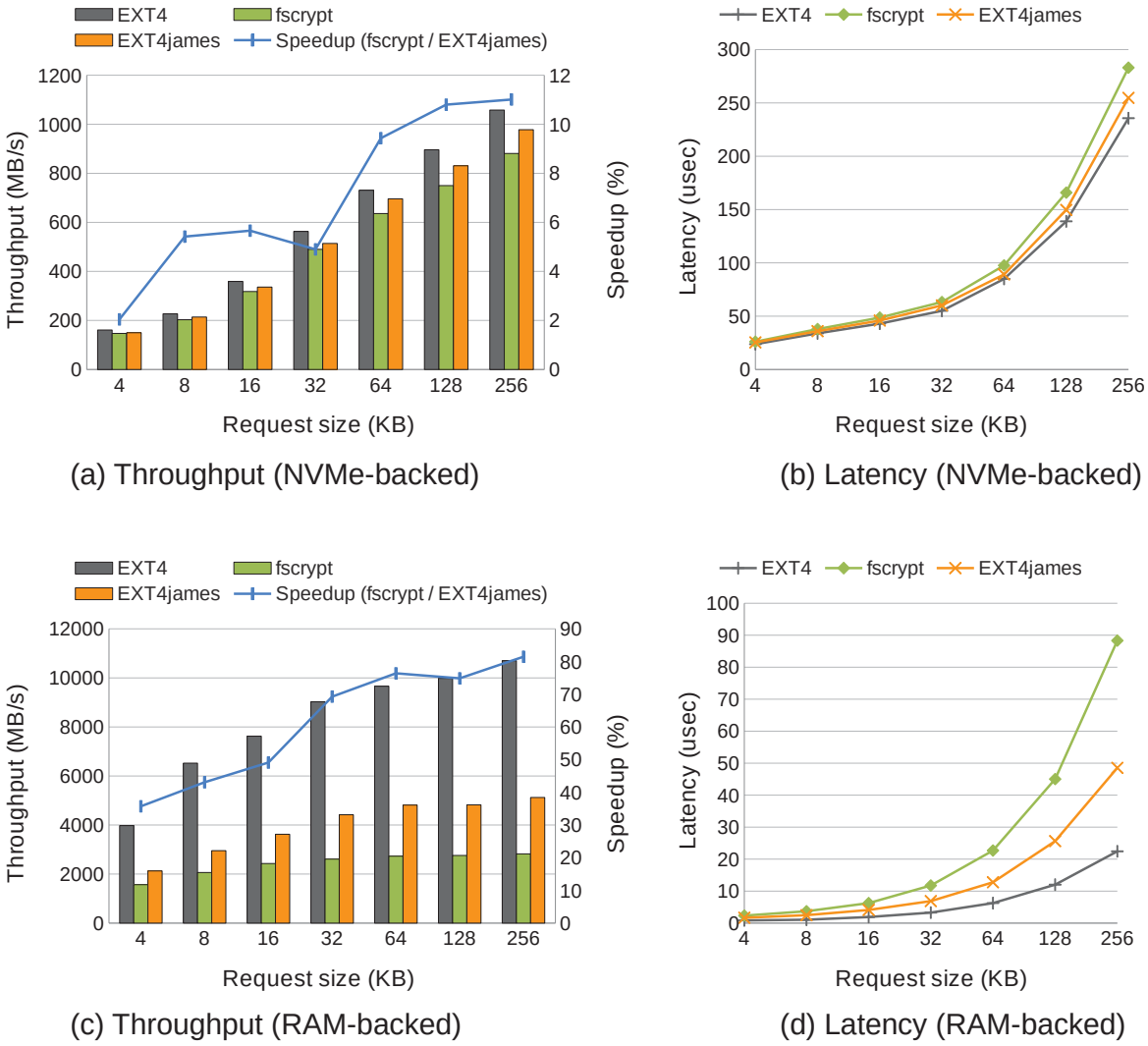



Figura 5.4: Escrita sequencial.

A figura 5.5 apresenta os resultados obtidos para a utilização das escritas randômicas. A vazão foi melhorada em 13% e 74% para dispositivos NVMe e baseados em RAM, respectivamente, e a latência foi melhorada em 12% e 43% respectivamente.

Assim como para as leituras, o padrão randômico gera operações de E/S maiores, diminuindo a importância as operações criptográficas no *overhead* final, e consequentemente, diminuindo o peso das otimizações realizadas pelo EXT4james. Contudo, vale ressaltar que a criação especulativa de máscaras para as operações de escritas não apresentou a perda de desempenho verificada nas requisições de leitura de 4KB para dispositivos baseados em RAM. Além do processamento especulativo, a máscara é aplicada pela própria *thread* cliente, dispensando a necessidade de trocas de contextos.

## 5.3 AMBIENTES SEM ACELERAÇÃO DE OPERAÇÕES CRIPTOGRÁFICAS

Sistemas de arquivos do Android utilizam fscrypt para realizar encriptação baseada em arquivos (Android Open Source Project, 2023). Nas plataformas móveis, assim como CPU virtuais e sistemas embarcados, é comum não existir suporte em hardware para aceleração de operações criptográficas como o AES-NI. Buscando analisar o desempenho da proposta

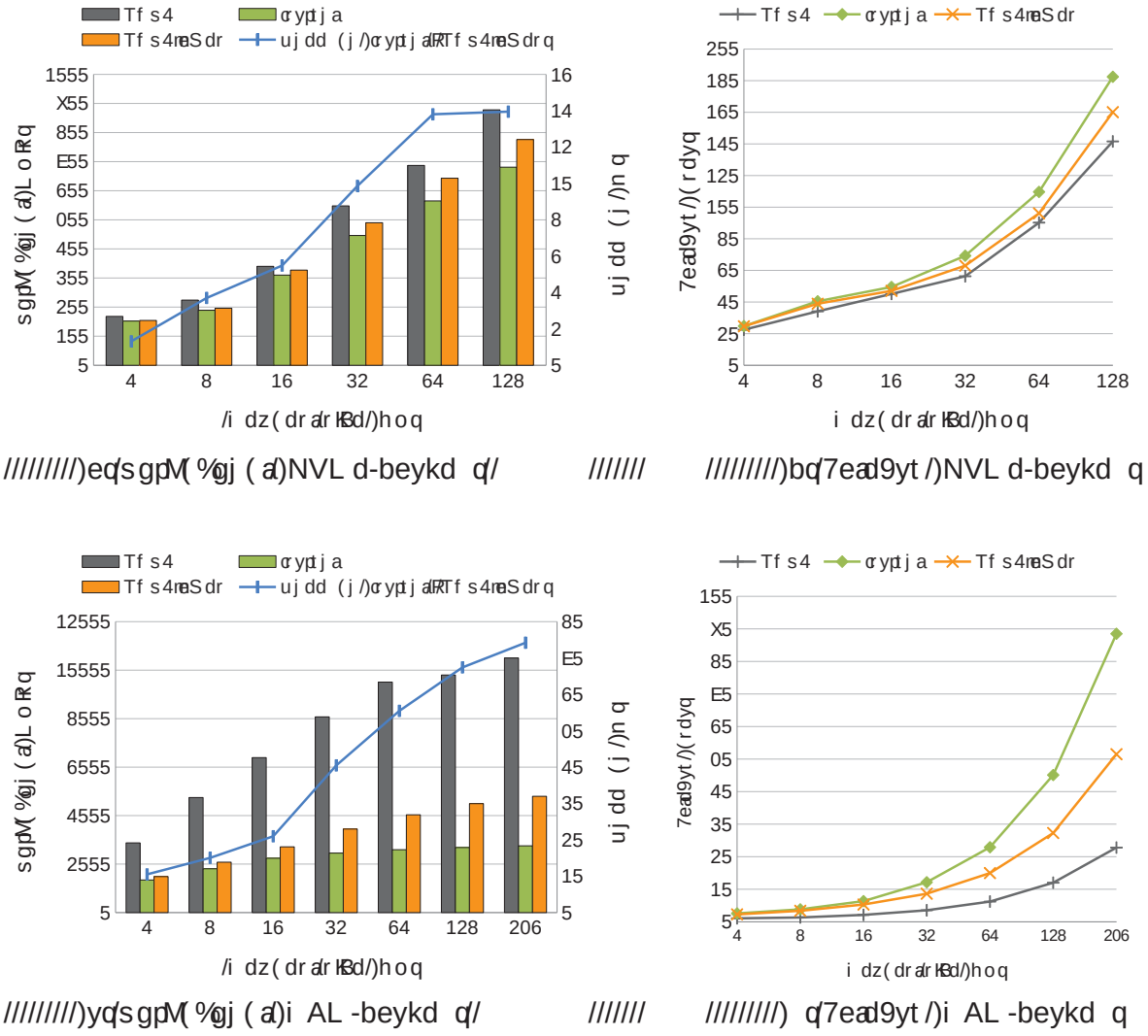



Figura 5.5: Escrita randômica.

nestes ambientes, foram executadas as cargas de trabalho anteriores com a função AES-NI desabilitada. As melhorias de vazão para dispositivos NVMe estão apresentadas na Tabela 5.4 e para dispositivos RAM na Tabela 5.5, enquanto as melhorias de latência para dispositivos NVMe estão apresentadas na Tabela 5.6 e para dispositivos RAM na tabela 5.7.

Neste contexto, o tempo de E/S se mantém como no contexto anterior, mas as operações criptográficas tomam mais tempo, fazendo com que esse *overhead* seja mais significativo no tempo total e possibilitando o EXT4james esconder a execução de mais instruções. Por esse motivo, o EXT4jmes consegue alcançar até 615% de aumento da vazão e diminuir até 86% da latência em operações de escritas em RAM por exemplo, (como mostrado nas Tabelas 5.5 e 5.7, respectivamente), quando comparado com o fscrypt.

## 5.4 LIMITAÇÕES DO AMBIENTE DE TESTE

A partir da análise dos resultados obtidos nas seções anteriores, percebe-se que o ganho de desempenho proporcionado pelo EXT4james é bem maior quando executado sobre um dispositivo de blocos baseado em RAM. Isso se dá pelo fato de que a latência do dispositivo NVMe utilizado e do barramento PCIe agem como um gargalo que aumenta o tempo das operações

Tabela 5.4: Porcentagem (%) de melhora da vazão do EXT4james em relação ao fscrypt com AES-NI desabilitado em dispositivos NVMe

| Request size (KB) | Sequential Read | Random Read | Sequential Write | Random Write |
|---|---|---|---|---|
| 4 | 50.9 | 13.4 | 51 | 61.7 |
| 8 | 91.1 | 19.5 | 65.9 | 59.2 |
| 16 | 160.4 | 37 | 117 | 100.6 |
| 32 | 245.2 | 54.3 | 158.2 | 143.5 |
| 64 | 344 | 105.3 | 255.8 | 228.9 |
| 128 | 461.2 | 168.7 | 249 | 254.2 |
| 256 | 501.4 | - | 253.7 | - |

Tabela 5.5: Porcentagem (%) de melhora da vazão do EXT4james em relação ao fscrypt com AES-NI desabilitado em dispositivos RAM

| Request size (KB) | Sequential Read | Random Read | Sequential Write | Random Write |
|---|---|---|---|---|
| 4 | 8.6 | 10.1 | 482.3 | 430.7 |
| 8 | 78.1 | 81.8 | 561 | 535.3 |
| 16 | 182.7 | 176.4 | 620.2 | 506.6 |
| 32 | 254.5 | 247.2 | 612.2 | 538.9 |
| 64 | 348.1 | 354.8 | 629.6 | 606 |
| 128 | 407.3 | 409.8 | 605.3 | 599.3 |
| 256 | 466.9 | 472 | 615 | 615 |

Tabela 5.6: Porcentagem (%) de melhora da latência do EXT4james em relação ao fscrypt com AES-NI desabilitado em dispositivos NVMe

| Request size (KB) | Sequential Read | Random Read | Sequential Write | Random Write |
|---|---|---|---|---|
| 4 | 34.4 | 12 | 33.7 | 39 |
| 8 | 48.2 | 16.6 | 40.2 | 61.3 |
| 16 | 62 | 27.1 | 54.2 | 51 |
| 32 | 71.3 | 35.2 | 61.5 | 59.3 |
| 64 | 77.7 | 51.5 | 72.0 | 70 |
| 128 | 82.2 | 62.9 | 71.4 | 72 |
| 256 | 83.4 | - | 71.8 | - |

Tabela 5.7: Porcentagem (%) de melhora da latência do EXT4james em relação ao fscrypt com AES-NI desabilitado em dispositivos RAM

| Request size (KB) | Sequential Read | Random Read | Sequential Write | Random Write |
|---|---|---|---|---|
| 4 | 7.2 | 8.4 | 84.2 | 66.3 |
| 8 | 44.0 | 45.6 | 85.5 | 71.1 |
| 16 | 65.2 | 64.5 | 86.7 | 85 |
| 32 | 72.2 | 71.8 | 86.3 | 85 |
| 64 | 78 | 78.4 | 86.5 | 86 |
| 128 | 80.4 | 80.6 | 86 | 86 |
| 256 | 82.4 | 82.7 | 86 | 85.8 |

de E/S, diminuindo o impacto das operações criptográficas no tempo total e aproximando o desempenho de todos os sistemas de arquivos testados. Contudo, o dispositivo baseado em RAM

não libera a CPU durante as operações de E/S como ocorre com o dispositivo NVMe, o que não permite simular o comportamento de um dispositivo de armazenamento para múltiplos clientes.

Além disso, as cargas de trabalho de leitura e escrita randômicas de 256KB executadas sobre o dispositivo NVMe apresentaram uma variação de desempenho significativa. Em uma única execução da carga de trabalho de leitura, por exemplo, a vazão variava entre 1MB e 400MB constantemente, resultando em uma vazão que não condiz com o cenário de execução. Esse comportamento se verificou no sistema de arquivos EXT4, mas como o fscrypt e o EXT4james utilizam o EXT4 como base, todos os sistemas de arquivos foram afetados, impossibilitando realizar qualquer comparação. Este comportamento também se repetiu em vários tamanhos de requisição nos testes de múltiplos clientes para as requisições de escrita. Tendo em vista as dificuldades desses cenários, é interessante o estudo aprofundado desses casos em trabalhos futuros.

## 5.5 MÚLTIPLOS CLIENTES

Visto que o SAC proposto realiza as operações criptográficas no nível de requisição do disco, não existe diferença entre o tratamento de requisições geradas por uma única *thread* cliente ou por clientes diferentes. Contudo, quanto mais *threads* clientes, mais ocupada estará a CPU, gerando menos tempo ocioso para geração de máscaras. Em contrapartida, quanto mais clientes, mais requisições de máscaras serão realizadas, gerando a agregação do tratamento pelas *threads* trabalhadoras. A Figura 5.6 apresenta os resultados obtidos através da aplicação das cargas de trabalho de leitura sequencial e randômica para 8 clientes, que é o número de núcleos de processamento da CPU utilizada.

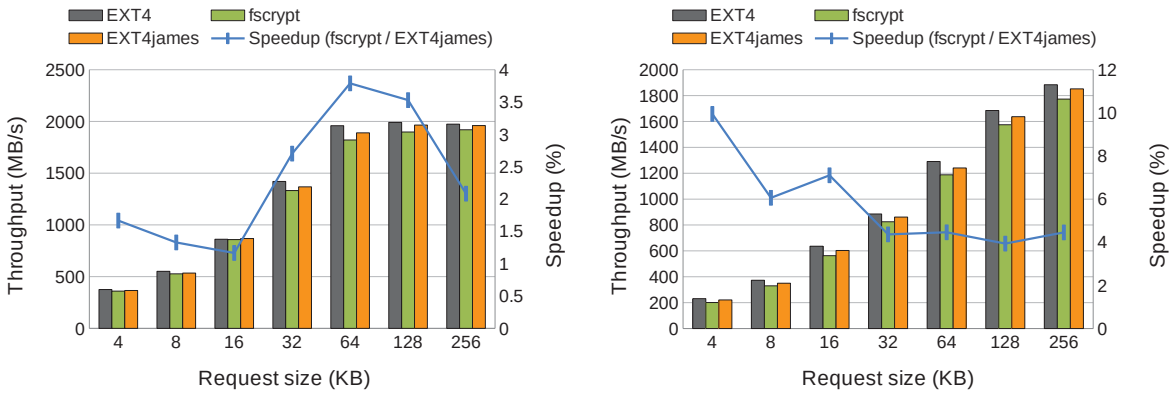



Figura 5.6: Cargas de leitura para múltiplos clientes.

Percebe-se que a vazão dos três sistemas de arquivos estão muito próximas, resultado do gargalo imposto pela largura de banda do dispositivo NVMe. Ainda assim, percebe-se que o EXT4james consegue aproveitar qualquer tempo ocioso da CPU para realizar o preparo das máscaras criptográficas e gerar pequenos ganhos de desempenho quando comparados com o fscrypt.

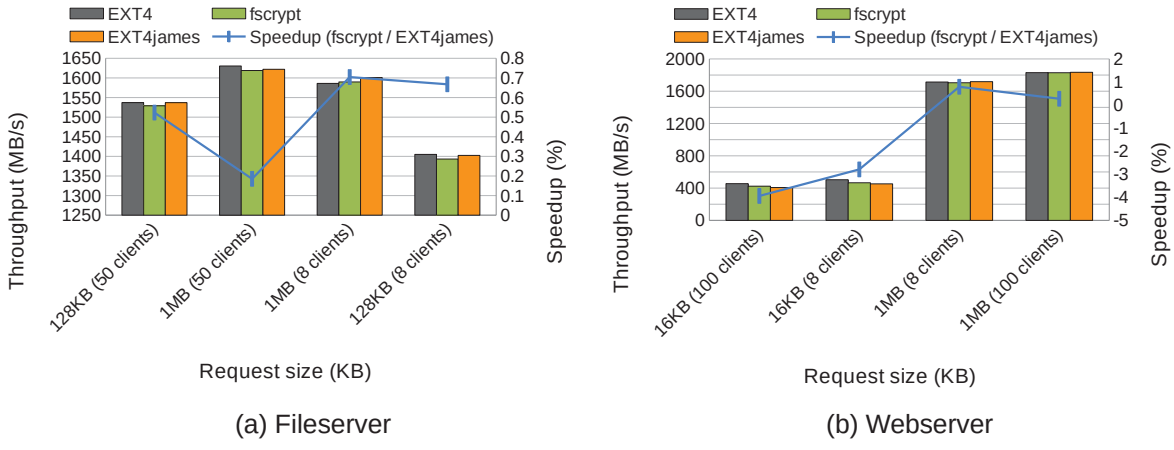
A proposta deste trabalho visa otimizar operações de criptografia em cenários onde a CPU fica ociosa por um tempo. Em cenários onde não existe ociosidade, o único *overhead* introduzido pela arquitetura proposta é a troca de contexto entre as *threads* clientes e as *threads* de *kernel*, visto que não existe execução de operações criptográficas adicionais em relação à aplicação do modo de operação CTR padrão.

Ainda que não seja possível fazer comparações de desempenho para as operações de escrita com múltiplos usuários, percebeu-se uma possível limitação na implementação do EXT4james: o acesso ao bloco de *nonce* em memória para as operações de escrita. Quando um *nonce* precisa ser escrito, a *thread* cliente é liberada, o bloco de *nonces* é bloquado para a escrita e o EXT4james dispara uma *thread* para realizar a escrita de forma assíncrona. Contudo, a existência de várias *threads* clientes operando sobre o mesmo arquivo de forma assíncrona faz com que cada requisição necessite da mudança total do bloco de *nonce*. Ainda que as *threads* clientes não sejam impactadas diretamente, o desempemho geral do sistema é reduzido. Uma solução seria manter um bloco de *nonce* para cada arquivo aberto, fazendo com que o padrão de acesso de um cliente não influencie o bloco mapeado por outro cliente. Porém, essa solução necessitaria também de um algoritmo de coerência de cache, pois um cliente pode modificar *nonces* mapeados por outro. Leituras assíncronas também sofrem com a mudança de blocos mapeados em memória, mas como mostraram os testes anteriores, o benefício da criação *ahead-of-time* supera o tempo de E/S do bloco de *nonce*. Porém, as operações de escrita precisam persistir o bloco atual e buscar o novo bloco, resultando em uma operação de E/S a mais.

## 5.6 CARGAS DE TRABALHO DE *MACRO-BENCHMARK*

Como mencionado anteriormente, algumas aplicações podem precisar ser adaptadas para solucionar o alinhamento mais restritivo requisitado pelo fscrypt. Esse é o caso do Filebench. Por isso, esta ferramenta foi modificada para que as operações *direct I/O* fossem sempre alinhadas ao tamanho da página, 4KB. Para exemplificar a utilização da proposta em aplicações reais, foram utilizadas as cargas de trabalho *fileserver* e *webserver*. A carga de trabalho *fileserver* realiza operações de criação, abertura, fechamento, escrita, leitura, remoção e requisições de metadados, objetivando simular o comportamento de um servidor de arquivos. A carga de trabalho *webserver* realiza operações de abertura, leitura, fechamento e escrita de arquivos, objetivando simular o comportamento de um servidor web.

A Figura 5.7 apresenta os resultados obtidos através da execução das cargas citadas anteriormente. Como esperado, a limitada largura de banda do dispositivo NVMe não nos possibilita analisar os ganhos de desempenho quando comparados com o fscrypt, contudo, é possível verificar o comportamento do EXT4james para os cenários em questão.




(a) Fileserver (b) Webserver

Figura 5.7: Cargas de trabalho de *macro-benchmark*.

Para a carga de trabalho *fileserver*, são utilizadas 50 *threads* clientes operando sobre um conjunto de 10000 arquivos de em média 128KB. Com o objetivo de analisar o comportamento

dos sistemas de arquivos, foram variados o tamanho médio dos arquivos para 1MB e a quantidade de clientes para o número de núcleos disponíveis. Percebe-se que mesmo com arquivos maiores e com 50 *threads* clientes, o EXT4james consegue manter a sua vazão limitada pela largura de banda do NVMe, mantendo a utilização da CPU pelas *threads* trabalhadoras em 25%, de forma a não gerar *overheads significativos*.

Para a carga de trabalho *webserver*, são utilizadas 100 *threads* clientes operando sobre um conjunto de 100 arquivos de em média 16KB. Como já verificado nos testes de *micro-benchmark*, tamanhos de arquivos pequenos tendem a proporcionar menos oportunidades de otimização, resultando em uma ligeira perda de desempenho. Já para arquivos de 1MB, ainda que com 100 clientes ocupando a CPU, o EXT4james não sofre com grandes perdas de desempenho e mantém o seu desempenho limitado pela largura de banda.

# 6 CONCLUSÃO

Este trabalho propôs um SAC *cross-stack* que tira vantagem de funções e informações disponíveis em cada camada da pilha de armazenamento do Linux para executar operações criptográficas de forma antecipada e especulativa. A avaliação de desempenho do protótipo implementado indica que a esta abordagem apresenta uma diminuição considerável na latência das operações, impactando também na vazão total de dados. Além disso, percebe-se que a utilização da WAESlib em espaço de *kernel*, gerenciamento dos seus contextos criptográficos e *nonces* são bem incorporadas à arquitetura geral do SAC proposto.

De forma geral, os maiores ganhos de desempenho encontrados se dão nos cenários onde existe um dispositivo de armazenamento rápido. Isso se dá pelo fato de que a porcentagem de tempo gasto pelas operações criptográficas é maior, de forma que a arquitetura proposta consegue esconder uma parte maior da latência total das requisições. Dessa forma, este trabalho apresenta uma solução que pode acompanhar o desenvolvimento dos dispositivos de armazenamento, tornando o processo de criptografia menos impactante a medida que a tecnologia de hardware avança.

É perceptível também que, quando as operações criptográficas são lentas, como é o caso dos cenários que não utilizam aceleração em hardware, as operações criptográficas também acabam ocupando uma porcentagem maior no tempo total das operações, criando mais oportunidades de otimização com a utilização da criptografia especulativa e *ahead-of-time*.

Outra variável importante que influencia no desempenho da arquitetura proposta é o tamanho das requisições. Quanto maior a requisição, mais máscaras criptográficas podem ser criadas durante a execução da operação de E/S, aumentando assim a utilização do tempo que seria perdido.

O tratamento de forma similar entre o padrão sequencial e randômico nas operações de leitura geraram ganhos de desempenho consideráveis, evitando a invalidação de máscaras já preparadas e diminuindo a complexidade do gerenciamento dos contextos de execução. De forma análoga, a criação de máscaras de forma especulativa proporciona ganhos de desempenhos nas operações de escrita, até em requisições pequenas.

A pilha de armazenamento do *kernel* Linux foi explorada de forma a tirar vantagem das funcionalidades específicas providas por cada camada. Durante a idealização da arquitetura do sistema, percebeu-se que o conceito *cross-stack* pode ser aplicado a diferentes pilhas de armazenamento, visto que o tempo gasto nas operações de E/S sempre vai existir, gerando oportunidades de otimizações.

Como trabalhos futuros, de planeja-se implementar mecanismos de tolerância a falhas, com o objetivo de manter a consistência entre dados e metadados em situações de falhas. Além disso, algoritmos mais sofisticados de cache de *nonces* podem diminuir a quantidade de operações de E/S de metadados, facilitar a forma de persistência dos mesmos e possivelmente integrar com os futuros mecanismos de tolerância a falhas. Também planeja-se associar mecanismos de gerenciamento de chaves, afim de permitir a encriptação de dados em diferentes granularidades e por diferentes usuários.

Além disso, pretende-se incrementar os testes com a utilização de um hardware mais eficiente, afim de verificar o real limite da arquitetura proposta. Associado a isso, serão executadas diferentes cargas de trabalhos em aplicações reais utilizando cargas de trabalho conhecidas como YCSB (Cooper et al., 2010) e TPC-H (Transaction Processing Performance Council, 2023), com o objetivo de prospectar novos cenários que apresentem ganhos de desempenho.

# REFERÊNCIAS


Android Open Source Project (2023). File-based encryption. https://source.android.com/docs/security/features/encryption/file-based. Acessado em 27/06/23.

Axboe, J. (2023). Flexible io tester (fio). https://github.com/axboe/fio. Acessado em 25/06/23.

Bovet, D. P. e Cesati, M. (2005). *Understanding the Linux Kernel: from I/O ports to process management*. "O'Reilly Media, Inc.".

Broz, M. (2023a). dm-crypt: Linux kernel device-mapper crypto target. https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt. Acessado em 25/06/23.

Broz, M. (2023b). Linux unified key setup. https://gitlab.com/cryptsetup/cryptsetup/. Acessado em 27/06/23.

Carrier, B. (2005). *File system forensic analysis*. Addison-Wesley Professional.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R. e Sears, R. (2010). Benchmarking cloud serving systems with ycsb. Em *Proceedings of the 1st ACM symposium on Cloud computing*, páginas 143–154.

Corbet, J. (2010). Dynamic writeback throttling. https://lwn.net/Articles/405076. Acessado em 12/06/2023.

Correia, J. P. e Zola, W. M. N. (2022). Gpuramfs: Um sistema de arquivos em espaço de usuário para armazenamento volátil em gpu. Em *Computer on the Beach 2022*, páginas 309–311.

Correia, J. P. e Zola, W. M. N. (2023). Uma proposta para sistemas de arquivos criptográficos em kernel space. Em *XXIII Escola Regional de Alto Desempenho da Região Sul*.

de Andrade, V. C. O. (2022). Aceleração da criptografia especulativa em cpus multicore e sua aplicação em sistemas de arquivos em espaço de usuário. Dissertação de Mestrado, Pós-Graduação em Informática - Universidade Federal do Paraná, Curitiba - PR.

de Andrade, V. C. O. e Zola, W. M. N. (2022). Aplicabilidade da encriptação especulativa em cpus multicore para sistema de arquivo no espaço de usuário. Em *Computer on the Beach 2022*, páginas 282–289, Itajaí - Brasil.

Dworkin, M. (2001). Recommendation for block cipher modes of operation. methods and techniques. Relatório técnico, National Inst of Standards and Technology Gaithersburg MD Computer security Div.

Dworkin, M. J., Barker, E. B., Nechvatal, J. R., Foti, J., Bassham, L. E., Roback, E. e Dray Jr, J. F. (2001). Advanced encryption standard (aes).

Eckert, C., Erhard, F. e Geiger, J. (2000). GSFS—a new group-aware cryptographic file system. Em *2IFIP International Information Security Conference*, páginas 221–230, Bostom.

Eduardo, V. (2019). Sistema de arquivos criptográfico com aceleração especulativa em gpu. Dissertação de Mestrado, Pós-Graduação em Informática - Universidade Federal do Paraná, Curitiba - PR.

Eduardo, V., de Bona, L. C. E. e Zola, W. M. N. (2019). Speculative encryption on GPU applied to cryptographic file systems. Em *17th USENIX Conference on File and Storage Technologies (FAST 19)*, páginas 93–105, Boston, MA. USENIX Association.

Esteves, T., Macedo, R., Faria, A., Portela, B., Paulo, J., Pereira, J. e Harnik, D. (2019). Trustfs: an sgx-enabled stackable file system framework. Em *2019 38th International Symposium on Reliable Distributed Systems Workshops (SRDSW)*, páginas 25–30.

Ferguson, N., Schneier, B. e Kohno, T. (2011). *Cryptography engineering: design principles and practical applications*. John Wiley & Sons.

GNU Project (2015). The gnu privacy guard. `https://gnupg.org/`. Acessado em 22/06/23.

Gough, V. (2017). Encfs: An encrypted filesystem for fuse.

Gough, V. (2020). Encfs: an encrypted filesystem for fuse. `https://github.com/vgough/encfs`. Acessado em 17/09/2022.

Gueron, S. (2010). Intel advanced encryption standard (aes) new instructions set.

Halcrow, M. A. (2005). ecryptfs: An enterprise-class encrypted filesystem for linux. Em *2005 Linux Symposium*, páginas 201–218, Ottawa - Canada.

Hwang, J., Vuppalapati, M., Peter, S. e Agarwal, R. (2021). Rearchitecting linux storage stack for $\mu$s latency and high throughput. Em *OSDI*, páginas 113–128.

ISO/IEC 11889-1:2015 (2015). Information technology — trusted platform module library — part 1: Architecture. `https://www.iso.org/standard/66510.html`. Acessado em 22/06/23.

Jin, G. e Bo, L. (2011). Design and implementation of a cryptographic file system for linux based on trusted computing platform. Em *2011 Fourth International Conference on Intelligent Computation Technology and Automation*, páginas 102–105.

Jun, L. e Huiping, Y. (2010). Trusted full disk encryption model based on tpm. Em *The 2nd International Conference on Information Science and Engineering*, páginas 1–4.

Kantee, A. (2007). puffs-pass-to-userspace framework file system. *AsiaBSDCon 2007*, páginas 29–42.

Kernel Development Community (2023a). dm-crypt. `https://www.kernel.org/doc/html/v6.1/admin-guide/device-mapper/dm-crypt.html`. Acessado em 22/06/23.

Kernel Development Community (2023b). Encrypted keys for the ecryptfs filesystem. `https://www.kernel.org/doc/html/v6.1/security/keys/ecryptfs.html`. Acessado em 22/06/23.

Kernel Development Community (2023c). ext4 data structures and algorithms. https://www.kernel.org/doc/html/v6.1/filesystems/ext4/index.html. Acessado em 05/06/2023.

Kernel Development Community (2023d). ext4 data structures and algorithms. https://www.kernel.org/doc/html/v6.1/filesystems/ext4/index.html. Acessado em 05/06/2023.

Kernel Development Community (2023e). ext4 general information. https://www.kernel.org/doc/html/v6.1/admin-guide/ext4.html. Acessado em 05/06/23.

Kernel Development Community (2023f). Filesystem-level encryption (fscrypt). https://www.kernel.org/doc/html/v6.1/filesystems/fscrypt.html. Acessado em 12/06/2023.

Kernel Development Community (2023g). Inline encryption. https://www.kernel.org/doc/html/v6.1/block/inline-encryption.html. Acessado em 25/06/23.

Konsor, P. (2011). Avoiding avx-sse transition penalties. https://www.intel.com/content/dam/develop/external/us/en/documents/11mc12-avoiding-2bavx-sse-2btransition-2bpenalties-2brh-2bfinal-809104.pdf. Acessado em 12/06/2023.

Kusswurm, D. (2022). *Modern parallel programming with C++ and Assembly language : X86 SIMD development using AVX, AVX2, and AVX-512*. Apress, United States, 1nd. edition.

Lee, J., Heo, J., Park, J., Cho, Y., Hong, J. e Park, M. s. (2007). Buffer cache level encryption for embedded secure operating system. Em *International Conference on Embedded and Ubiquitous Computing*, páginas 346–355.

Leibenger, D., Fortmann, J. e Sorge, C. (2016). Encfs goes multi-user: Adding access control to an encrypted file system. Em *24th International Symposium on Research in Attacks, Intrusions and Defenses*, páginas 525–533.

Levin, D. V. e Mráz, T. (2023). Linux pam (pluggable authentication modules for linux) project. https://github.com/linux-pam/linux-pam. Acessado em 22/06/23.

Lipmaa, H., Rogaway, P. e Wagner, D. (2000). Ctr-mode encryption. Em *First NIST Workshop on Modes of Operation*, volume 39. Citeseer. MD.

Lomont, C. (2011). Introduction to intel advanced vector extensions. *Intel white paper*, 23.

Love, R. (2010). *Linux Kernel Development*. Pearson.

Martin, L. (2010). Xts: A mode of aes for encrypting hard disks. *IEEE Security & Privacy*, 8(3):68–69.

Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A. e Vivier, L. (2007). The new ext4 filesystem: current status and future plans. Em *Proceedings of the Linux symposium*, páginas 21–33.

Menezes, A. J., Van Oorschot, P. C. e Vanstone, S. A. (2018). *Handbook of applied cryptography*. CRC press.

Mueller, S. e Vasut, M. (2023). Linux kernel crypto api. https://www.kernel.org/doc/html/v6.1/crypto/index.html. Acessado em 05/06/2023.

National Institute of Standards and Technolog (2023). Special publication (sp) 800-38e: Recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38e.pdf. Acessado em 25/06/23.

Nickolls, J., Buck, I., Garland, M. e Skadron, K. (2008). Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53.

Nie, T. e Zhang, T. (2009). A study of des and blowfish encryption algorithm. Em *TENCON 2009 - 2009 IEEE Region 10 Conference*, páginas 1–4.

OpenSSL Comitters (2023). Openssl. https://www.openssl.org/. Acessado em 22/06/23.

Paar, C. e Pelzl, J. (2009). *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media.

Pontes, R., Burihabwa, D., Maia, F., Paulo, J., Schiavoni, V., Felber, P., Mercier, H. e Oliveira, R. (2017). Safefs: A modular architecture for secure user-space file systems: One fuse to rule them all. Em *Proceedings of the 10th ACM International Systems and Storage Conference*, páginas 1–12.

Rawat, U. e Kumar, S. (2012). Ecfs: An enterprise-class cryptographic file system for linux. *International Journal of Information Security and Privacy (IJISP)*, 6(2):53–63.

Reinders, J. R. (2013). Intel® avx-512 instructions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html. Acessado em 12/06/2023.

Rott, J. K. (2012). Intel® advanced encryption standard instructions (aes-ni). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html. Acessado em 12/06/2023.

Ruan, Z., He, T. e Cong, J. (2019). Insider: Designing in-storage computing system for emerging high-performance drive. Em *USENIX Annual Technical Conference*, páginas 379–394.

Schmuck, F. B. e Haskin, R. L. (2002). Gpfs: A shared-disk file system for large computing clusters. Em *FAST*, volume 2.

Sebastien, G. (2023). sysstat - system performance tools for the linux operating system. https://github.com/sysstat/sysstat. Acessado em 26/06/23.

Shannon, C. E. (1949). Communication theory of secrecy systems. *The Bell system technical journal*, 28(4):656–715.

Sharma, R., Kallurkar, P., Kumar, S. e Sarangi, S. R. (2012). Usife: a user space file system with support for intra-file encryption. Em *Fourth International Conference on Machine Vision (ICMV 2011): Machine Vision, Image Processing, and Pattern Analysis*, páginas 321–325.

Sharma, S., Moona, R. e Sanghi, D. (2006). Transcrypt: A secure and transparent encrypting file system for enterprises. Em *8th International Symposium on System and Information Security*.

Stallings, W. (2017). *Cryptography and Network Security: Principles and Practice*. Pearson.

Steere, D. C., Kistler, J. J. e Satyanarayanan, M. (1990). *Efficient user-level file cache management on the Sun vnode interface*. Carnegie-Mellon University. Department of Computer Science.

Sun, W., Ricci, R. e Curry, M. J. (2012). Harnessing GPU computing for storage systems in the OS kernel. Em *Proceedings of the Fifth International Systems and Storage Conference (SYSTOR)*.

Szeredi, M. (2003). Filesystem in USEr space. *http://sourceforge. net/projects/avf*.

Tanenbaum, A. S. (2014). *Modern Operating Systems*. JPearson, 4 edition.

Transaction Processing Performance Council (2023). Tpc-h. `https://www.tpc.org/tpch/`. Acessado em 27/06/23.

Van Tilborg, H. C. e Jajodia, S. (2014). *Encyclopedia of cryptography and security*. Springer Science & Business Media.

Vangoor, B. K. R., Tarasov, V. e Zadok, E. (2017). To FUSE or not to FUSE: Performance of User-Space file systems. Em *15th USENIX Conference on File and Storage Technologies (FAST 17)*, páginas 59–72, Santa Clara, CA. USENIX Association.

Vasily, T. (2016). Filebench: A flexible framework for file system benchmarking.; login. *The Usenix Magazine*, 41:6.

Wang, L., Wen, Y., Kong, J. e Yi, X. (2012). Optimizing ecryptfs for better performance and security. Em *Linux Symposium*, página 137, Ottawa - Canada.

Westerlund, A. e Danielsson, J. (1998). Arla: a free afs client. Em *Proceedings of the annual conference on USENIX Annual Technical Conference*, páginas 32–32.

Wilcox, M. (2018). The xarray data structure. `https://lwn.net/Articles/745073/`. Acessado em 18/09/2022.

Xiao, C., Pan, Y., Xu, D., Liu, W., Sun, S. e Qiu, S. (2020). Load-aware adaptive cache management scheme for enterprise-level stackable cryptographic file system. Em *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, páginas 35–43.

Xiao, C., Zhang, L., Liu, W., Cheng, L., Li, P., Pan, Y. e Bergmann, N. (2018). Nv-ecryptfs: accelerating enterprise-level cryptographic file system with non-volatile memory. *IEEE Transactions on Computers*, 68(9):1338–1352.

Yang, Z., Harris, J. R., Walker, B., Verkamp, D., Liu, C., Chang, C., Cao, G., Stern, J., Verma, V. e Paul, L. E. (2017). Spdk: A development kit to build high performance storage applications. Em *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, páginas 154–161.

Zadok, E., Badulescu, I. e Shender, A. (1998). Cryptfs: A stackable vnode level encryption file system. Relatório técnico, Technical Report CUCS-021-98, Computer Science Department, Columbia University.

Zhang, I., Liu, J., Austin, A., Roberts, M. L. e Badam, A. (2019). I'm not dead yet! the role of the operating system in a kernel-bypass era. Em *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, página 73–80, New York, NY, USA. Association for Computing Machinery.

Zhong, Y., Li, H., Wu, Y. J., Zarkadas, I., Tao, J., Mesterhazy, E., Makris, M., Yang, J., Tai, A., Stutsman, R. e Cidon, A. (2022). XRP: In-Kernel storage functions with eBPF. Em *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, páginas 375–393, Carlsbad, CA. USENIX Association.

Zola, W. M. N. e De Bona, L. C. E. (2012). Parallel speculative encryption of multiple aes contexts on gpus. Em *2012 Innovative Parallel Computing (InPar)*, páginas 1–9. IEEE.