Criptografia Antecipada Aplicada a Sistemas de Arquivos de Baixa Latência

Jorge Pires Correia[1], **Wagner Machado N. Zola**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

{jpcorreia, wagner}@inf.ufpr.br

Abstract. *As storage devices evolve, the time spent by software to read or write data becomes increasingly relevant. This characteristic is accentuated in the context of cryptographic file systems, where the software layer is in charge of performing cryptographic operations. Seeking to address this problem, this work proposes a solution that hides the latency of cryptographic operations in low-latency environments through ahead-of-time encryption, taking advantage of low-level interfaces of the Operating System kernel.*

Resumo. *À medida que os dispositivos de armazenamento se desenvolvem, o tempo gasto em software para ler ou escrever dados se torna cada vez mais relevante. Essa característica se acentua no contexto de sistemas de arquivos criptográficos, onde a camada de software é encarregada de realizar operações criptográficas. Buscando atacar este problema, o presente trabalho propõe uma solução que esconde a latência das operações criptográficas em ambientes de baixa latência através da criptografia antecipada, tirando proveito de interfaces de baixo nível do núcleo do Sistema Operacional.*

1. Introdução

Podemos dividir a latência de uma requisição de leitura ou escrita de dados a um dispositivo de armazenamento em duas etapas: latência de software e latência de hardware. A latência de software é composta pelo tempo gasto pelo código do Sistema Operacional (SO), o qual realiza operações de gerenciamento e validações de segurança. Após o SO repassar a requisição para o dispositivo de armazenamento, a latência de hardware entra em cena, de modo que o dispositivo demora um certo tempo para realizar a operação requisitada.

A utilização de dispositivos de armazenamento mecânicos baseados em discos magnéticos (HDDs) implica em uma latência de hardware ordens de magnitude maior que a latência de software, de modo que a latência de hardware representa quase 100% da latência total da requisição. Contudo, graças a tecnologias de memórias *flash* presentes em dispositivos como SSDs e NVMes, os dispositivos de armazenamento se desenvolveram ao ponto que a latência de hardware pode representar somente 50% da latência total [Zhong et al. 2022].

Sistemas de Arquivos Criptográficos (SACs) aumentam a latência de software em requisições de leitura e escrita de dados através da execução de operações de criptografia, visando manter a privacidade e a confidencialidade dos dados nos dispositivos de armazenamento. Considerando a utilização de dispositivos como HDDs, o aumento da latência

imposto pelos SACs é irrelevante, tendo em vista que a latência de hardware é bastante superior. Contudo, o contrário é observado quando dispositivos de baixa latência são utilizados. Nesse sentido, a otimização de operações criptográficas tem alto potencial de melhora do desempenho total dos sistemas.

Conceitualmente, as operações criptográficas em um SAC são executadas em sequência com as operações usuais. Em uma requisição de leitura, o SAC precisa esperar que o dispositivo de armazenamento coloque o dado na memória principal (RAM) antes de iniciar o processo de decriptação. De maneira similar, em operações de escrita, o dispositivo de armazenamento precisa esperar que o SAC realize as operações de encriptação para que o dado possa ser armazenado. A necessidade da execução sequencial dos passos apresentados anteriormente decorre da forma com que os algoritmos criptográficos funcionam. Contudo, o algoritmo Warped AES (WAES) [Zola and De Bona 2012] introduziu um método para a execução do algoritmo criptográfico *Advanced Encryption Standard* (AES), usando o modo de operação *Counter* (CTR) de forma que a execução das operações criptográficas mais custosas é realizada de forma **antecipada** e **paralela**. Essas duas características do WAES se encaixam perfeitamente no contexto de SACs: o processamento antecipado pode permitir que as operações criptográficas custosas sejam realizadas antes que o dado esteja pronto na memória RAM; o processamento paralelo pode permitir que requisições que sejam feitas em um mesmo momento sejam tratadas em diferentes núcleos de processamento, sem qualquer restrição.

Essa ideia foi explorada pelo EncFS++ [Eduardo et al. 2019], um SAC que executa em espaço de usuário e que foi capaz de melhorar a vazão e a latência quando comparado com o EncFS [Gough 2017], um SAC convencional, que também executa em espaço de usuário, e que serviu de base para a implementação do EncFS++. Contudo, a utilização de um sistema de arquivos de espaço de usuário implica em uma degradação da latência [Vangoor et al. 2017], tendo em vista que o sistema de arquivos de usuário precisa se comunicar com o *driver* responsável pela interface com os mecanismos internos do núcleo do SO. Esse problema é acentuado caso um dispositivo de armazenamento de baixa latência seja utilizado. Para exemplificar esse comportamento, a sobrecarga de desempenho imposta pelo FUSE [Vangoor et al. 2017], uma biblioteca popular para a implementação de sistemas de arquivos em espaço de usuário, pode chegar a $2\times$, quando utilizado um dispositivo de armazenamento NVMe (demonstrado na Seção 4). Dessa maneira, a sobrecarga de desempenho citada anteriormente não é aplicável em um contexto de baixa latência proporcionada pelos dispositivos de armazenamentos atuais.

Embora exista um potencial na aplicação de criptografia antecipada em sistemas de arquivos, como demonstrado pelo EncFS++, um *design* que suporte este tipo de sistema carece de pesquisa. Por isso, uma solução que combine de forma eficiente esses conceitos dentro do núcleo do SO pode superar a fraqueza que implementações em espaço de usuário possuem. Tendo em vista que um SAC que utiliza operações de criptografia antecipada precisa lidar com fluxos de execuções e metadados bastante singulares, nós propomos uma nova arquitetura de SAC que tira proveito de interfaces internas do núcleo do SO para executar operações criptográficas de forma otimizada. Para validar nossa proposta, nós implementamos uma prova de conceito, através da adaptação do sistema de arquivos padrão do Linux, o ext4, e outras camadas da pilha de armazenamento. O resultado apresentou uma melhora significativa na latência e vazão de dados para operações de

leitura e escrita quando comparado com outros SACs atuais.

Em resumo, esse artigo apresenta as seguintes contribuições: (i) demonstração do impacto de SACs em um contexto de baixa latência de hardware; (ii) identificação de componentes-chave para um SAC de criptografia antecipada; (iii) proposta de um novo *design* de SACs de criptografia antecipada, ressaltando como os componentes-chave podem ser integrados na pilha de protocolos de armazenamento do Linux; (iv) uma análise experimental da nossa proposta sobre diferentes cargas de trabalho.

2. Fundamentação Teórica

Nessa seção, nós apresentamos o contexto do subsistema de armazenamento do Linux utilizado como base para a integração do nosso SAC. Nós apresentamos também noções básicas sobre a criptografia antecipada.

2.1. Subsistema de Armazenamento do Linux

O gerenciamento de arquivos no Linux é composto por diferentes camadas de componentes. As chamadas de sistemas relacionadas a sistemas de arquivos são inicialmente recebidas pelo *Virtual File System* (VFS), que age como uma interface entre o espaço de usuário e as camadas inferiores. Após o recebimento e tratamento da chamada de sistema, o VFS identifica qual o sistema de arquivos responsável pelo arquivo que está relacionado com a chamada de sistema, e realiza o repasse da requisição. Sistemas de arquivos compõem a camada de software responsável pela política de acesso e organização dos dados do usuário nos dispositivos de armazenamento. O sistema de arquivos lê e escreve dados no dispositivo de armazenamento em granularidade de blocos de 4096 bytes, objetivando desempenho. Para realizar o acesso ao dispositivo de armazenamento, o sistema de arquivos realiza uma requisição para uma interface genérica denominada Camada de E/S de Blocos. Essa, por sua vez, recebe requisições em um formato padronizado e realizada requisições diretamente aos *drivers* dos dispositivos de armazenamento.

Existem algumas requisições de leituras e escritas que podem não precisar gerar operações de E/S para o dispositivo de armazenamento graças à *Page Cache*, um conjunto de páginas na memória RAM que mantém blocos (de 4096 bytes) do disco em cache. Quando uma requisição de leitura é recebida e o bloco que contém o dado está presente na *Page Cache*, a operação de leitura se resume a uma cópia de memória da *Page Cache* para o *buffer* do usuário, evitando a latência de hardware. Operações de escrita que modificam dados presentes na *Page Cache* se resumem a uma cópia de memória do *buffer* do usuário para a *Page Cache*, de forma que a persistência dos dados é agendada para um momento mais oportuno pelo núcleo do SO. Embora a *Page Cache* seja utilizada por padrão no Linux (em operações denominadas *buffered I/O*), algumas aplicações como Banco de Dados podem preferir utilizar políticas de cache próprias, devido aos seus padrões de acesso exclusivos. Por isso, o Linux oferece um tipo de requisição chamada *direct I/O*, que desativa a utilização da *Page Cache* e sempre gera requisições para a camada de E/S de blocos. Nesse caso, os dados são movidos diretamente entre o *buffer* do usuário e o dispositivo de armazenamento.

2.2. Criptografia Antecipada

Após a padronização do algoritmo Advanced Encryption Standard (AES) pelo NIST [Dworkin et al. 2001], este se tornou a especificação de criptografia mais utilizada em

diversos cenários. Por ser uma cifra de bloco, o AES realiza a criptografia em pequenas porções de dados de 128 bits. Tendo em vista que a maioria dos dados gerenciados por aplicações são maiores que o tamanho do bloco processado pelo AES, modos de operação como o *Cipher Block Chaining* (CBC), *XEX-based Tweaked-codebook mode with Ciphertext Stealing* (XTS) e *Counter* (CTR) surgiram, trazendo formas de aplicação do algoritmo AES em dados de tamanhos arbitrários.

O modo de operação CTR, apresentado na Figura 1, trata um conjunto de dados arbitrários como um conjunto de blocos de 128 bits. A cifra de blocos é aplicada sobre um contador (*counter*) de 128 bits, e o resultado é submetido a uma operação de XOR com o texto a ser encriptado/decriptado. O contador é normalmente concatenado com um Vetor de Inicialização (VI) aleatório, produzindo um valor utilizado somente uma vez (*nonce*) para uma mesma chave. O modo de operação CTR pode encriptar ou decriptar blocos de dados diferentes de forma paralela. A segurança desse modo de operação é comprovada, desde que seja seguida a forma de execução [Lipmaa et al. 2000], fazendo com que seja utilizada em diversos cenários.

A carga de trabalho criptográfica pesada é realizada na região pontilhada da Figura 1, de forma que os parâmetros necessários são o ***nonce*** e a **chave criptográfica**. O *insight* da criptografia antecipada é que o dado a ser encriptado/decriptado não é necessário nessa etapa. Este primeiro processamento gera uma **máscara criptográfica**, que será aplicada ao dado por meio de uma operação XOR, gerando o dado encriptado ou decriptado. Esse esquema, combinado com a possibilidade de criar máscaras criptográficas simultaneamente em diferentes núcleos de processamento, é explorado pelo WAES e implementado em uma biblioteca denominada WAESlib [Zola and De Bona 2012].

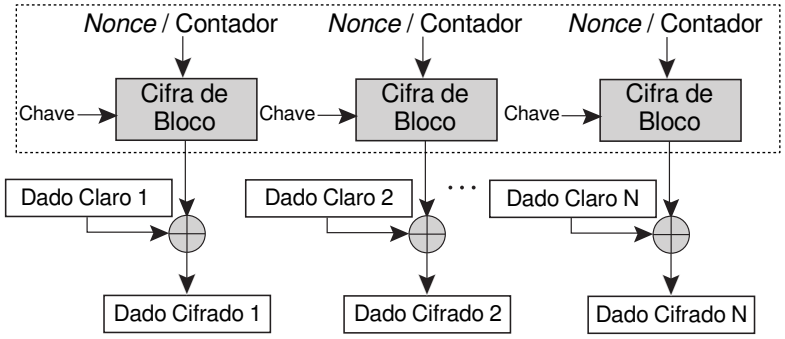

Figura 1. Modo de Operação CTR

3. Arquitetura e Implementação de um SAC Utilizando Criptografia Antecipada

A Figura 2a apresenta a execução de uma operação de leitura em um SAC convencional. Mesmo que a CPU esteja livre para executar qualquer tipo de trabalho enquanto a operação de E/S está sendo tratada pelo dispositivo de armazenamento, todas as operações devem ser executadas em sequência devido à necessidade de o dado estar em memória para ser processado. Por outro lado, através da aplicação da criptografia antecipada, a dependência do dado se restringe à mínima operação do XOR, e o tempo de CPU, que anteriormente era ocioso, pode ser utilizado para produzir máscaras criptográficas. Essa abordagem esconde a latência de operações de criptografias, como mostrado na Figura 2b. Se, para as operações de leitura, é possível criar máscaras enquanto as operações de E/S

estão sendo executas, máscaras para as operações de escrita podem ser criadas ainda mais cedo. Isso é possível porque novos blocos de dados utilizarão novos *nonces*, permitindo a criação de um conjunto de máscaras que servirão escritas futuras.

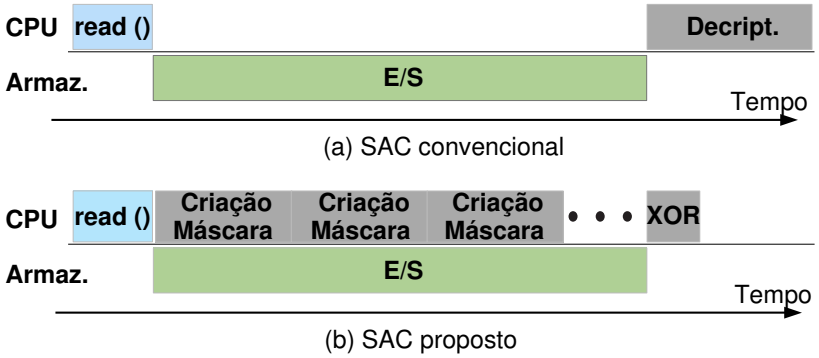

Figura 2. Execução de um SAC convencional e do SAC proposto

O modo de operação CTR possui a desvantagem de necessitar do armazenamento de um *nonce* para cada bloco de dados. Visto que o sistema de arquivos também lida com dados em uma granularidade de blocos, é possível armazenar um *nonce* para cada bloco de dados de 4096 bytes, reservando os bits menos significativos para servirem como um contador, incrementado a cada bloco de 128 bits. Dessa maneira, nós mantemos a segurança do modo de operação, além de manter a sobrecarga de espaço abaixo de 0.4%. Além disso, cada *nonce* é composto de duas partes: um VI imprevisível e um contador. Para cada requisição de escrita (criação de um novo bloco de dados de 4096 bytes ou modificação de um bloco existente), um VI imprevisível de 64 bits é utilizado com um contador de 64 bits, o qual é incrementado a cada bloco escrito.

Nossa proposta, denominada **ext4james**, busca manter a confidencialidade de dados que estão armazenados no dispositivo de armazenamento, de forma que confidencialidade de dados em memória está fora do escopo deste trabalho. Nós assumimos que o núcleo do SO é seguro. Por consequência, confiamos na integridade da biblioteca criptográfica provida pelo núcleo do Linux *Crypto API* [Mueller and Vasut 2025], tendo em vista que a utilizamos para realizar as operações criptográficas. Ainda que o gerenciamento de chaves criptográficas esteja fora do escopo deste trabalho, é possível incorporar diferentes soluções na nossa proposta através da *Crypto API*.

Durante a fase de idealização deste trabalho, foi possível identificar quatro desafios-chave que guiaram os esforços de implementação: armazenamento de *nonces*, fluxos de execução, adaptação da WAESlib e gerenciamento de contextos criptográficos.

3.1. Armazenamento de *Nonces*

Visto que um *nonce* de 16 bytes é necessário para o processo de decriptação de um bloco de dados, o seu armazenamento deve ser realizado de modo a permitir que a sua leitura e escrita não gere sobrecarga de desempenho. Por isso, foi adotado um esquema inspirado no gerenciamento de volumes do ext4 [Kernel Development Community 2025a], em conjunto com o conceito de *n-nodes* introduzido no EncFS++, além da utilização de interfaces providas pela camada de VFS e da camada de sistema de arquivos.

Os primeiros *nonces* de todos os arquivos de dados são armazenados em um único arquivo, denominado *Arquivo Global*. Caso seja necessário, cada arquivo de dados terá

um arquivo de armazenamento de *nonces* dedicado para armazenar os *nonces* que não puderam ser armazenados no Arquivo Global. Visando tirar máxima vantagem dos limites que o sistema de arquivos ext4 proporciona para os arquivos, nós organizamos o Arquivo Global conforme apresentado na Figura 3.

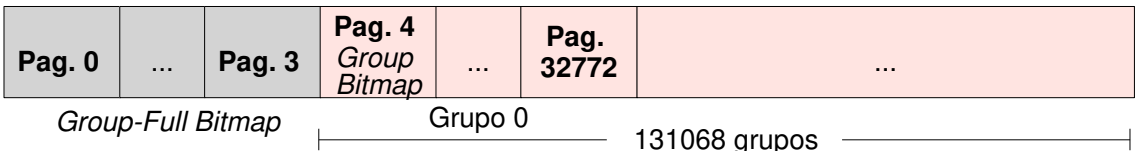

Figura 3. Gerenciamento do Arquivo Global

Os primeiros 256 *nonces* associados a um arquivo de dados, utilizados para decriptador os primeiros 1MB de dados, serão armazenados em uma página (bloco de 4096 bytes) do Arquivo Global. Este arquivo, por sua vez, é dividido em grupos, objetivando a facilidade no gerenciamento. Cada grupo possui 2^{15} + 1 páginas, onde a primeira página armazena o *Group bitmap* que controla as páginas ocupadas do grupo em questão. Todas as outras páginas de um grupo são destinadas a armazenar os *nonces* iniciais de um arquivo de dados.

As primeiras quatro páginas do Arquivo Global compõem o *Group-Full bitmap*. Este *bitmap* indica os grupos que todas as páginas estão armazenando *nonces* de algum arquivo, ou seja, não possuem mais espaços livres. A associação entre o *Group-Full bitmap* e os *group bitmaps* compõem um *bitmap* de dois níveis. Essa abordagem hierarquizada provê boa utilização do tamanho do Arquivo Global e acessos otimizados ao mesmo. Para endereçar uma página dentro do Arquivo Global, um inteiro de 32 bits pode ser utilizado: os primeiros 17 bits endereçam o grupo e os últimos 15 bits endereçam a página dentro do grupo.

Para associar um índice do Arquivo Global (e o arquivo de *nonces* dedicado, caso necessário) ao arquivo de dados, nós tiramos vantagem da camada de Sistema de Arquivos. O EXT4 provê um mecanismo chamado *Extended Attributes*, nos permitindo armazenar atributos customizáveis juntamente com os metadados do arquivo, sem a necessidade de utilização de blocos extras de armazenamento.

Para realizar a criação, deleção, leitura e escrita do Arquivo Global e dos arquivos dedicados de *nonces*, nós tiramos proveito das interfaces oferecidas pela camada do VFS, fazendo com que as operações citadas anteriormente ”pulem” as operações criptográficas e não gerem qualquer degradação de desempenho adicional. Para evitar a sobrecarga de software a cada acesso em um bloco de dados, nós realizamos a leitura e escrita de *nonces* em granularidade de blocos, mantendo um bloco de *nonces* em cache para cada arquivo de dados, de forma que a requisição de *nonces* em sequência não impacte o desempenho do SAC.

A geração de novos *nonces* ocorre em cada operação de escrita de dados. Nós mantemos um número inteiro global denominado *Contador Global*, que é incrementado em unidades de 256 (deixando 8 bits reservados para a divisão de um bloco de 4096 bytes em blocos menores de 128 bits/16 bytes) a cada requisição de escrita de dados. Então, quando uma operação de escrita é realizada, o valor atual do Contador Global é concatenado com um valor aleatório imprevisível de 64 bits, criando o *nonce* utilizado para realizar a encriptação do bloco de dados sendo escrito.

3.2. Fluxos de Execução

A Figura 4 apresenta os fluxos de execução do sistema proposto. As linhas vermelhas (pontilhadas) estão relacionadas ao fluxo executado nas operações de escrita (encriptação de dados), enquanto as linhas azuis (tracejadas) são executadas em operações de leitura (decriptação de dados). Linhas pretas (cheias) podem ser executadas para ambas operações. A comunicação entre a camada de sistemas de arquivos e a camada de E/S de blocos é realizada de forma padronizada através de uma estrutura denominada *BIO*, a qual contém informações como ponteiros para os blocos de dados e informações de identificação para acesso ao dado após a operação do dispositivo de armazenamento ser finalizada.

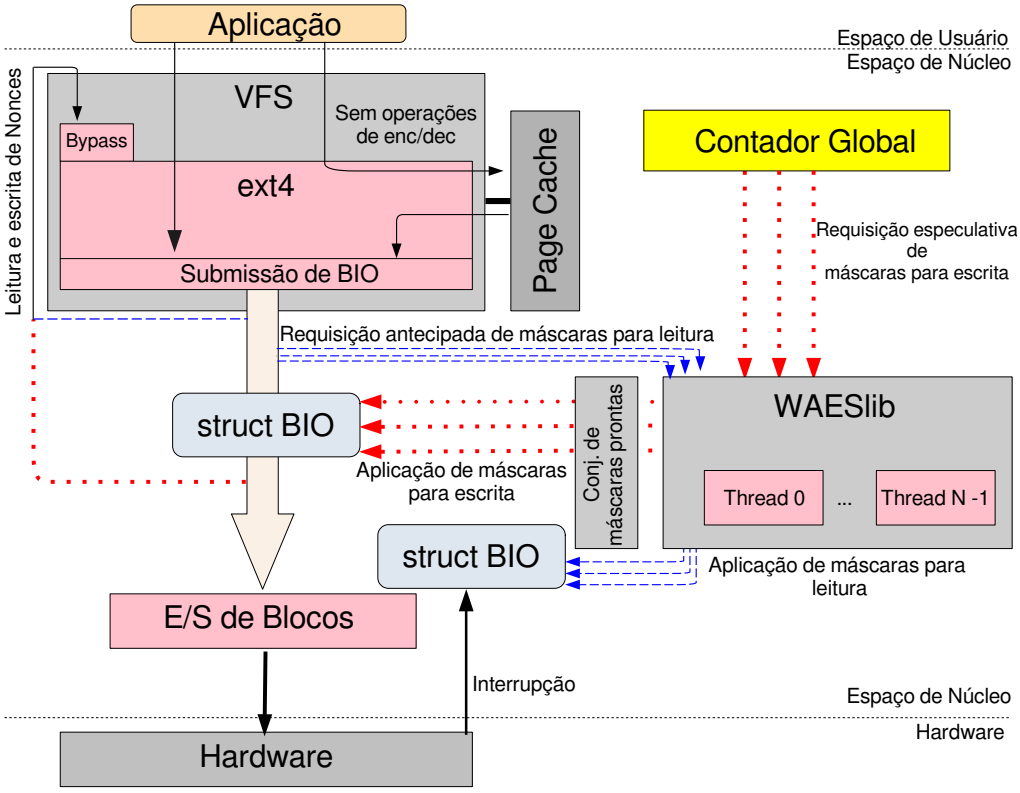

Figura 4. Arquitetura do SAC proposto

Podemos considerar que a operação mais importante em um SAC de criptografia antecipada é a criação de máscaras. O SAC precisa realizar a requisição de criação de máscaras em um momento tal que seja muito provável que a máscara esteja pronta para uso quando o dado estiver pronto em memória. Analisaremos, inicialmente, as operações de escritas. Requisições que modificam dados que estão armazenados na *Page Cache* devem ser ignorados pelas operações de criptografia, pois a memória deve permanecer decriptada e acessos a ela não devem impor degradação de desempenho. No instante que o SO decidir que é um momento oportuno de escrever páginas da *Page Cache* no disco, ou quando uma aplicação realizar operações de escrita do tipo *Direct I/O*, o SAC já precisa ter máscaras criptográficas prontas para aplicação. Isso significa que as requisições de criação de máscaras devem ser feitas antes mesmo das requisições de escritas existirem. Por isso, foi utilizada uma abordagem especulativa de criação de máscaras criptográficas para escrita. Como novas escritas utilizarão novos *nonces*, é possível criar um conjunto de

máscaras que serão consumidas a cada requisição feita. Dessa forma, é bastante provável que exista uma máscara pronta quando uma requisição de escrita for executada.

Assim como em operações de escrita, operações de leitura que acessam dados que estão presentes na *Page Cache* não devem gerar degradação de desempenho. Quando uma página precisar ser buscada do dispositivo de armazenamento para a *Page Cache*, ou a aplicação executar uma requisição de leitura do tipo *direct I/O*, o sistema de arquivos cria uma estrutura *BIO* que contém os blocos que deverão ser lidos e o *buffer* que receberá os dados, e a submete para a camada de E/S de blocos. Diferente das operações de escrita, a decriptação de dados acontece após o término da operação do dispositivo de armazenamento. Dessa forma, o tempo da latência de hardware pode ser aproveitado para criar as máscaras criptográficas. Quando a estrutura *BIO* está pronta, logo antes da submissão para a camada de E/S de blocos, o SAC já tem acesso aos índices dos blocos envolvidos na operação. Com esta informação, é possível encontrar os *nonces* referentes aos blocos no Arquivo Global ou no arquivo de *nonces* dedicado, e submeter a requisição de criação de máscaras. Quando o dispositivo de armazenamento gerar uma interrupção, indicando a finalização da operação de leitura, as máscaras, muito provavelmente, estarão prontas para serem aplicadas sobre o dado.

3.3. Adaptação da WAESlib

A WAESlib é responsável por prover criptografia antecipada por meio de uma interface simples. Visto que a WAESlib é uma biblioteca de espaço de usuário, se fez necessária uma completa reimplementação para permitir o seu uso no contexto do núcleo do SO. A criação de máscaras, que na implementação original era realizada com o suporte da GPU, foi reimplementada utilizando AES-NI [Gueron 2010], um conjunto de instruções presente nos processadores atuais que proporciona baixa latência na execução de operações criptográficas. A biblioteca Crypto API [Mueller and Vasut 2025], provida pelo núcleo do Linux, foi utilizada para executar operações AES-NI de forma transparente. As operações de XOR foram executadas utilizando AVX [Lomont 2011], um conjunto de instruções que utilizam registradores vetoriais para lidar com dados em palavras grandes.

Tendo em vista que as requisições de criação de máscaras podem ser tratadas de forma paralela, foi criado um conjunto de *threads* trabalhadoras, as quais permanecem bloqueadas e acordam se existirem requisições de criação pendentes. As requisições são enfileiradas em uma *heap* binária, permitindo a implementação de um mecanismo de prioridades para as máscaras criadas.

3.4. Gerenciamento dos Contextos Criptográficos

Um contexto criptográfico é um conceito provido pela WAESlib, o qual define a estrutura que representa uma operação criptográfica para um bloco. Cada contexto contém um *buffer* de origem de dados, um *buffer* de destino, uma chave criptográfica, um *nonce* e uma prioridade. Dessa forma, para a encriptação ou decriptação de um bloco de dados, se faz necessário reservar um contexto criptográfico, preenchê-lo com as informações necessárias e submetê-lo para WAESlib.

A WAESlib provê todos os contextos criptográficos individualmente, sendo obrigação do SAC gerenciá-los. Como as máscaras criptográficas das operações de escrita são criadas em um contexto de execução diferente que as máscaras utilizadas em operações de escrita, o gerenciamento delas é realizado de forma diferente.

Primeiramente, iremos analisar o gerenciamento dos contextos criptográficos para as operações de leitura. Cada requisição de leitura é composta por um conjunto de blocos (4096 bytes) sequenciais. Visto que cada bloco precisa de uma máscara, um conjunto de contextos criptográficos é necessário para cara requisição. O cenário ideal seria reservar um número de contextos criptográficos igual ao número de blocos sendo lidos, o que permitiria que as *threads* trabalhadoras produzissem o maior número de máscaras possível. Contudo, um conjunto de processos realizando grandes requisições de leitura poderia reservar inúmeros contextos, desbalanceando a utilização entre o restante dos processos. Por isso, nós definimos um número máximo de contextos reservados por requisição. Se um processo está lendo mais blocos que o tamanho do conjunto de contextos, o SAC tratará aquele conjunto como uma janela deslizante, de forma que quando uma máscara for aplicada sobre um dado, o contexto desta máscara será usado para criar a próxima máscara ainda não utilizada. O número máximo de contextos por requisições foi implementado de forma variável, podendo ser modificado de acordo à carga de trabalho do sistema: um tamanho maior de conjunto de contextos pode causar maior uso de memória, por outro lado, um menor tamanho do conjunto de páginas pode evitar que *threads* trabalhadoras produzam todas as máscaras necessárias durante o tempo da latência de hardware.

Para as requisições de escrita, foi definido um único conjunto de contextos de tamanho fixo. Esse conjunto de contexto é tratado como uma janela deslizante que cria máscaras criptográficas, utilizando o Contador Global. A criação de novas máscaras se inicia no momento de montagem do SAC. Quando uma operação de escrita acontece, o SAC pode aplicar uma máscara já computada, de forma que o *nonce* utilizado para a criação da máscara será armazenado no Arquivo Global ou em um arquivo dedicado de *nonces*. Após a utilização da máscara, o contexto que continha a máscara em questão pode ser reutilizado para a criação de uma nova máscara que será utilizada em operações de escritas futuras.

4. Resultados e Discussões

Nós executamos nossos testes utilizando a ferramenta de benchmark FIO [Axboe 2025], que nos permite criar altas cargas de trabalho para sistemas de arquivos. Os testes foram executados na nossa versão modificadao do 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 DDR5, e um dispositivo de armazenamento NVMe ADATA SX6000LNP. As políticas do *Linux Governor*, que gerencia a escala de frequência da CPU, foram definidas para o nível de *performance*. Todas as requisições do usuário foram executadas através de operações *Direct I/O*, sempre gerando requisições para o sistema de arquivos.

Visto que o EncFS++ não provê suporte a requisições de *Direct I/O*, nós decidimos executar o sistema de arquivos BBFS [Pfeiffer 2018] como base de desempenho de sistemas de arquivos em espaço de usuário. BBFS é um sistema de arquivos que executa em espaço de usuário através da biblioteca FUSE (mesma biblioteca utilizada pelo EncFS e EncFS++) que simplesmente repassa todas as requisições para o ext4. Como o EncFS++ realiza as operações de criptografia antes de repassar as requisições para o ext4, podemos assumir que o BBFS é sempre mais rápido que o EncFS++.

Considerando que nosso objetivo é demonstrar a vantagem do nosso sistema,

ext4james, com relação aos SAC atuais, nós comparamos nossa execução com o SAC *fscrypt* [Kernel Development Community 2025b], um SAC implementado no núcleo do SO que também utiliza ext4 como base. Além disso, o fscrypt utiliza o modo de operação XTS para aplicação da criptografia AES através da Crypto API. As semelhanças entre o fscrypt e o ext4james facilitam as comparações e evidenciam as vantagens da arquitetura proposta. Buscanto uma comparação justa, nós também executamos uma versão do ext4james que utiliza o modo de operação CTR sem a criptografia antecipada (CTR padrão), facilitando o entendimento das otimizações propostas. Nós também executamos testes com o sistema de arquivos ext4 puro, sem qualquer operação criptográfica, a fim de prover um parâmetro de comparação para os SACs, já que todos utilizam o ext4 como base.

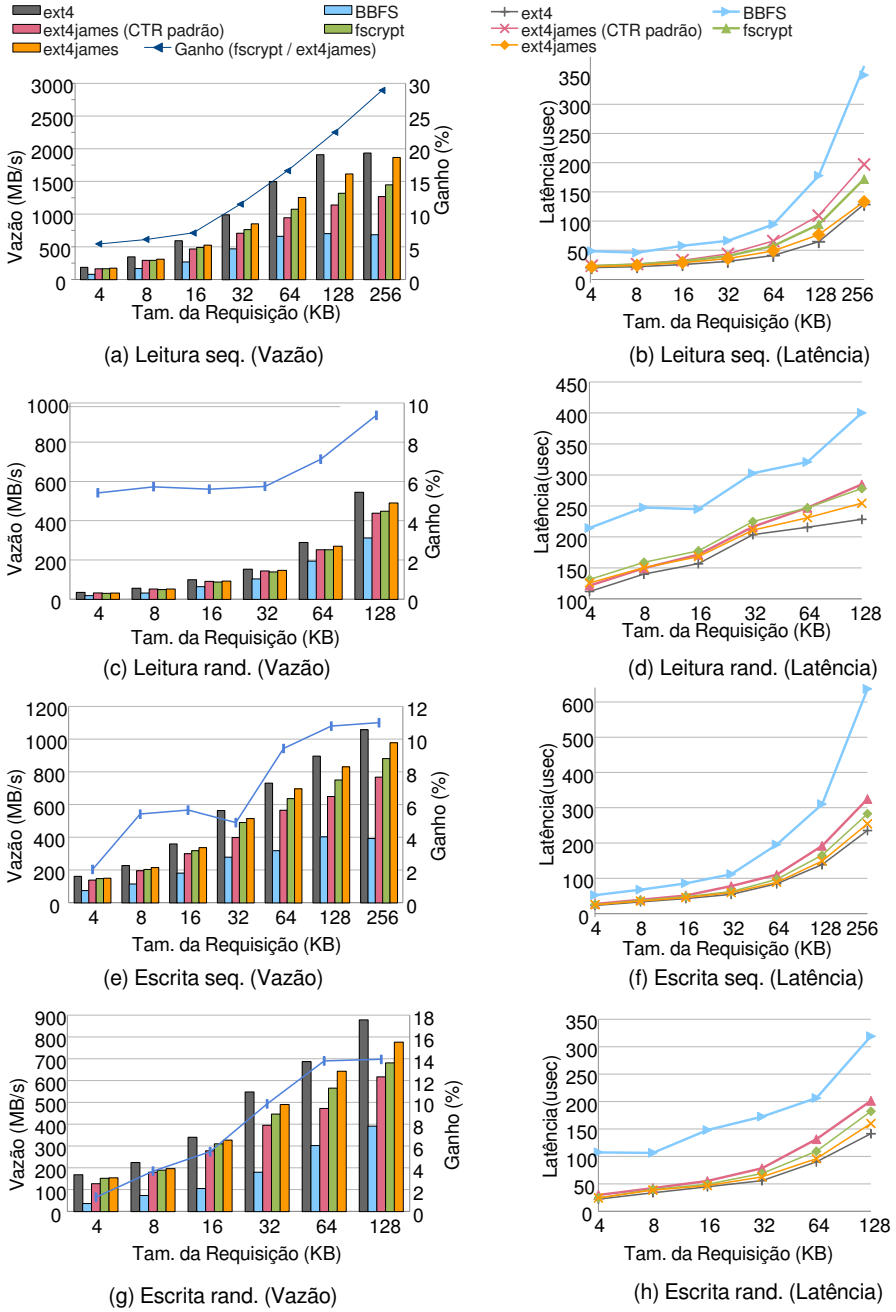

Figura 5. Cargas de Trabalho Sequenciais e Aleatórias de Leitura e Escrita

A Figura 5a apresenta a vazão de leituras sequencias utilizando o dispositivo de armazenamento NVMe. No eixo Y direito, é apresentado o ganho, em porcentagem, do ext4james em comparação com o fscrypt. A Figura 5b apresenta a latência para as mesmas operações. O primeiro ponto a ser considerado é que somente utilizar o modo de operação CTR não gera melhoras no desempenho. O modo de operação XTS, que é otimizado para utilização em armazenamento de dados, provê ganhos de desempenho quando comparado com o modo de operação CTR puro. Contudo, ao ser utilizada a criptografia antecipada, o ext4james melhora a vazão em até 28% e a latência em até 22%. Vale ressaltar, também, que ainda que o BBFS não execute qualquer operação criptográfica, a latência apresentada é maior que todos os SACs apresentados, demonstrando a degradação de desempenho imposta por uma execução em espaço de usuário. Ademais, a vazão e latência do ext4james se aproximam do ext4 a medida que o tamanho das requisições aumentam, fazendo com que a latência das operações criptográficas fiquem quase que totalmente escondidas. As Figuras 5c e 5d apresentam o desempenho dos sistemas em uma carga de trabalho de leituras aleatórias. O resultado mostram uma melhora no desempenho da vazão em até 9% e melhora na latência em até 8%. Visto que o dispositivo de armazenamento leva mais tempo para realizar acessos aleatórios, a latência de software se torna menos representativa, escondendo as otimizações das operações criptográficas. Nós decidimos não apresentar os testes feitos para as requisições aleatórias de 256KB, pois para o sistema de arquivos ext4 (e consequentemente para todos os outros SACs), sofreu com grandes variações nos tempos de requisição, impossibilitando uma avaliação dos tempos de criptografia. Acreditamos que esse comportamento está relacionado ao hardware utilizado, mas consideramos que uma análise mais aprofundada seja necessária em trabalhos futuros.

As Figuras 5e e 5f apresentam os resultados para as cargas de trabalho de escritas sequenciais. Estes resultados mostram uma melhora na vazão de até 11% e na latência de até 9%. Embora os ganhos de desempenho apresentados para as cargas de trabalho de escrita sejam menores que as apresentadas para a leitura, nós mantivemos a latência muito próxima da latência do ext4 , indicando que o gargalo não está nas operações criptográficas, mas sim na latência de hardware do NVMe. As Figuras 5g e 5h apresentam comportamentos similares encontrados nas escritas aleatórias, onde nossa proposta melhora a vazão em até 13% e a latência em até 12%.

5. Conclusão

Este trabalho propôs um SAC que tira proveito das interfaces e informações presentes em diferentes camadas do núcleo do SO para prover criptografia antecipada em ambientes de baixa latência. Os resultados dos testes da nossa prova de conceito mostraram uma melhora de desempenho para diferentes padrões de acesso quando comparado com arquiteturas convencionais. Os resultados também contribuíram para a discussão sobre comportamentos dos SACs. Além disso, nossa proposta tem potencial de prover melhores desempenhos a medida que os dispositivos de armazenamento evoluem, visto que a latência introduzida pela camada de criptografia se torna cada vez mais relevante. Para trabalhos futuros, nós planejamos a integração de soluções de gerenciamento de chaves criptográficas para maior usabilidade. Ademais, planejamos a integração com soluções de gerenciamento de metadados para controle de tolerância a falhas, buscando incorporar soluções já existentes na literatura [Liu et al. 2018], as quais controlam metadados de aplicações que utilizam o modo de operação AES-CTR.

Agradecimentos

Este trabalho teve apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Referências

Axboe, J. (2025). Flexible io tester (fio). `https://github.com/axboe/fio`.

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

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

Gough, V. (2017). EncFS: An encrypted filesystem for FUSE.

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

Kernel Development Community (2025a). ext4 data structures and algorithms. `https://www.kernel.org/doc/html/v6.1/filesystems/ext4/index.html`.

Kernel Development Community (2025b). Filesystem-level encryption (fscrypt). `https://www.kernel.org/doc/html/v6.1/filesystems/fscrypt.html`.

Lipmaa, H., Rogaway, P., and Wagner, D. (2000). Comments to NIST concerning AES modes of operations: CTR-mode encryption. In *First NIST Workshop on Modes of Operation*, volume 39. Citeseer. MD.

Liu, S., Kolli, A., Ren, J., and Khan, S. (2018). Crash consistency in encrypted non-volatile main memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 310–323. IEEE.

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

Mueller, S. and Vasut, M. (2025). Linux kernel crypto API. `https://www.kernel.org/doc/html/v6.1/crypto/index.html`.

Pfeiffer, J. J. (2018). Writing a fuse filesystem: a tutorial. `https://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/`.

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

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

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