PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA POLITÉCNICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ESTEGANOGRAFIA EM SISTEMA DE ARQUIVOS DO SISTEMA OPERACIONAL LINUX

GABRIEL OLIVEIRA LAUREANO

GOIÂNIA
2021

GABRIEL OLIVEIRA LAUREANO

ESTEGANOGRAFIA EM SISTEMA DE ARQUIVOS DO SISTEMA OPERACIONAL LINUX

Trabalho de Conclusão de Curso apresentado à Escola Politécnica, da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Orientador:

Prof. Olegário Correa da Silva Neto.

Banca examinadora:

Prof. Dr. Fábio Barbosa Rodrigues

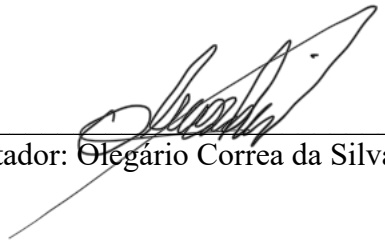
Prof. Dr. José Luiz de Freitas Júnior

GOIÂNIA
2021

GABRIEL OLIVEIRA LAUREANO

ESTEGANOGRAFIA EM SISTEMA DE ARQUIVOS DO SISTEMA OPERACIONAL LINUX

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola Politécnica, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel em Ciência da Computação, em 09 / 12 / 2021.



Orientador: Olegário Correa da Silva Neto

Prof. Ma. Ludmilla Reis Pinheiro dos Santos

Coordenadora de Trabalho de Conclusão de Curso

GOIÂNIA
2021

Aos meus familiares, amigos e docentes pelo apoio,
Que em meu quadro de depressão,
Me incentivaram a continuar e retomar o foco.

AGRADECIMENTOS

Aos professores da banca de avaliação, José Luiz de Freitas Júnior e Fábio Barbosa Rodrigues pela compreensão e correções pontuais em momentos complicados.

À Coordenação da Escola Politécnica da Pontifícia Universidade Católica de Goiás por ajudar na execução deste projeto.

Aos demais docentes pelos conselhos e sugestões durante meu período de graduação.

Aos meus colegas de graduação, aos quais viemos a formar um time coeso para estudo e desenvolvimento de projetos.

Ao professor Olegário Correa da Silva Neto, orientador acadêmico, pelo apoio, paciência e confiança, que possibilitou a realização deste trabalho.

“De fato, agimos da mesma forma quando estamos acordados.”

Friedrich Nietzsche, Crepúsculo dos Ídolos, 1900

RESUMO

Com o avanço dos ataques computacionais para roubo de informações, a natureza da segurança sobre dados está sempre em foco. O uso de sistemas de arquivos pela maioria dos computadores pessoais cria a possibilidade de que formas eficientes de ocultação e manipulação desta estrutura sejam relevantes. No presente trabalho, realizou-se um estudo do sistema de arquivos do sistema operacional Linux, sua arquitetura e comunicação com o núcleo do sistema operacional, como uma análise de métodos de esteganografia que utilizem essa estrutura.

Palavras-Chave: *Sistema de Arquivos, Esteganografia, EXT4, FUSE.*

ABSTRACT

With the advancement of computer attacks to steal information, the nature of data security is always in focus. The use of file systems by most personal computers creates the possibility that efficient ways of hiding and manipulating this structure are relevant. This work presents studies about file system of the Linux operating system, architecture and communication with the operating system kernel, as an analysis of steganography methods that use this structure.

Keywords*: Filesystem, Steganography, EXT4, FUSE.*

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

ABNT	Associação Brasileira de Normas Técnicas
CFM	*Common File Model*
CI	*Integrated Circuits*
CLI	*Command Line Interface* — Interface de Linha de Comandos
EXT2	*Second Extended File System*
EXT3	*Third Extended File System*
EXT4	*Fourth Extended File System*
FMS	*Fortran Monitor System*
FS	*File System*
FUSE	*Filesystem in Userspace*
IBSYS	*IBM Operating System*
ID	*Identifier*
I/O	*Input/Output*
NFS	*Network File System*
NFTS	*New Technology File System*
OS	*Operating System*
OS/360	*IBM System/360 Operating*
PID	*Process Identifier*
POSIX	*Portable Operating System Interface*
StegFS	*Steganographic File System*
SYSFS	*System File System*
VFS	*Virtual File System*

SUMÁRIO

CAPÍTULO I – INTRODUÇÃO

Com a criação e aperfeiçoamento de novos métodos de ataques computacionais para roubo de informações, a natureza da segurança sobre dados importantes está sempre em foco. O uso de sistemas de arquivos pela maioria dos computadores pessoais cria a necessidade de que novas formas de ocultação e manipulação dados nestes sistemas sejam estudadas, visando prevenir ataques obsoletos.

Um estudo aprimorado dos sistemas de arquivos que documenta de forma expositiva pontos cruciais do funcionamento pode levar a implementação de novos métodos de esteganografia, condicionando um melhor entendimento dessas estruturas ou lidando com situações singulares durante a execução de testes, colocando em discussão funcionalidades ainda não exploradas.

1.1 Objetivos

Este trabalho possui os seguintes objetivos:

- Apresentar um estudo sobre o sistema operacional e suas diferentes relações do kernel com os sistemas de arquivos;
- Apresentar um estudo sobre a estrutura do sistema de arquivos, com foco no sistema EXT4 e FUSE;
- Analisar e especular sobre possíveis métodos de esteganografia com foco em sistemas de arquivos.

1.2 Justificativa

Esse tema foi escolhido devido ao fato de a complexidade do sistema de arquivos ser pouco explorada durante o uso de técnicas de esteganografia. Abrir a possibilidade para um novo uso da esteganografia em conjunto com a estrutura de dados desses sistemas gera novas discussões e especulações sobre novos métodos. Ao longo de gerações de desenvolvimento dos sistemas de arquivos, novas formas de compactar e endereçar bytes em condições seguras e eficientes vêm sendo implementadas, deixando várias minúcias pelo caminho que raramente são documentadas. Assim, colocar ênfase neste assunto direciona o sistema de arquivos para ser uma camada a mais de proteção contra ataques futuros.

1.3 Metodologia

No decorrer do trabalho, foram utilizados como materiais computadores que utilizam sistema operacional com base Linux. Os estudos foram realizados através de artigos, livros, apostilas e

documentação de software. Testes pontuais para verificação de mecânicas dos sistemas de arquivos foram realizadas em terminal *shell bash* 5.0.17.

1.4 Organização Do Trabalho

Este trabalho é dividido em seis capítulos. Neste primeiro capítulo foram apresentadas as motivações as quais fundamentaram o estudo de sistemas de arquivos.

No segundo capítulo são apresentados conceitos básicos do sistema operacional Linux, como seu histórico e relações pontuais do kernel com sistema de arquivos.

No terceiro capítulo é abordado um estudo do sistema de arquivos e sua estrutura em ambiente Linux, com foco no VFS, FUSE, EXT4 e seu sistema de *journaling*.

No quarto capítulo é abordado um estudo fundamental sobre esteganografia, histórico de criação, conceitos iniciais e estratégias pontuais previamente utilizadas.

No quinto capítulo são analisados os métodos e possíveis técnicas para o uso de esteganografia em nível de sistema de arquivos.

No sexto são apresentadas as considerações finais, dificuldades encontradas e sugestões para trabalhos futuros.

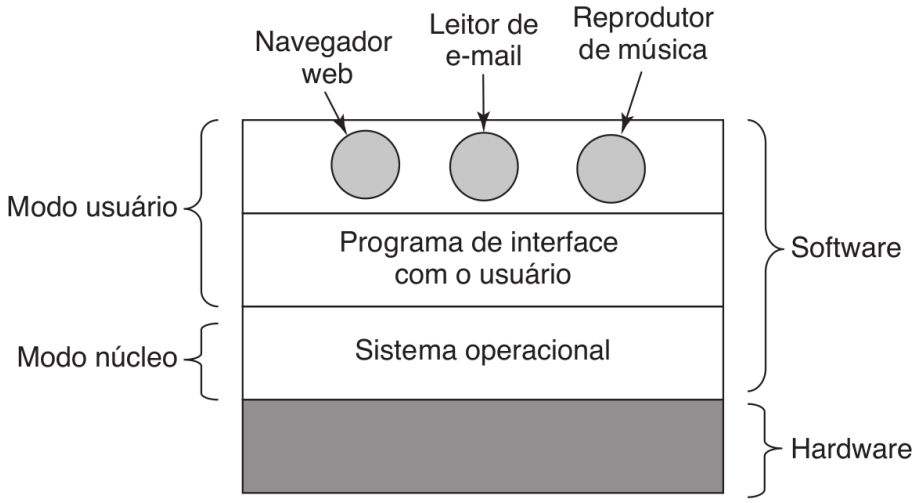
CAPÍTULO II – SISTEMA OPERACIONAL LINUX

Neste capítulo são abordados, de forma essencial, os conceitos estruturais do sistema operacional e suas relações com o sistema de arquivos, focando nas características do Sistema Operacional Linux.

2.1 Introdução

Conforme Tanenbaum (2016), o computador é um conjunto de componentes, como processadores, monitores, teclado, unidades de armazenamento, e dispositivos de entrada e saída. Para agir de forma coordenada e eficiente, é usado um software chamado sistema operacional. Este software provê ao usuário um gerenciamento de recursos como processos e armazenamento, criando uma ponte de interação entre o hardware, software e usuário. Hierarquia simplificada dos principais componentes de um computador é ilustrada na Figura 1.

Figura 1 - Onde o sistema operacional se encaixa.



Fonte: TANENBAUM, 2016.

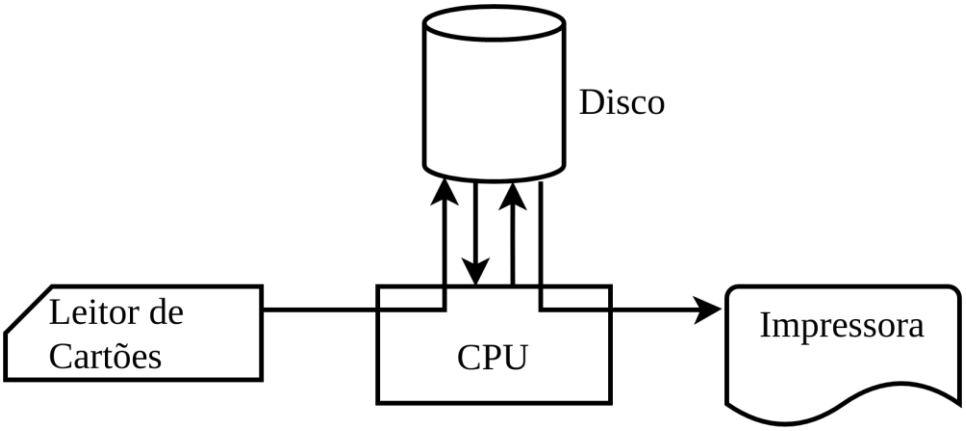
2.2 Histórico

Conforme Tanenbaum (2016), o sistema operacional no início esteve muito ligado à como o hardware funcionava, sendo dificilmente tratado de forma separada até o nascimento da terceira geração de computadores, quando já contemplava o uso de ICs (*Integrated Circuits*, Circuitos integrados).

Foi na tentativa de criação de um sistema único que fosse compatível com todos os modelos de computadores em produção na época que o termo sistema operacional criou popularidade, sendo que o sistema FMS (*Fortran Monitor System*, Sistema de Monitoramento por Fortran) e o IBSYS (*IBM Operating System*, Sistema Operacional IBM) ainda estavam conceitualmente ligados às máquinas físicas.

A criação deste sistema único, OS/360 (*IBM System/360 Operating System*, Sistema Operacional System/360 IBM), seguiu em desenvolvimento e teve seu lançamento em 1966, falhando em conseguir ser realmente compatível com todas as máquinas da época de forma eficiente, mas abrindo portas para novas técnicas que seriam necessárias nos sistemas operacionais para a evolução da computação. Destas técnicas se destaca o *Spooling* (do acrônimo, *Simultaneous Peripheral Operation On-Line*), Figura 2, que consistia na utilização do armazenamento em disco para agilizar o carregamento de tarefas a serem executadas (SILBERSCHATZ, 2015).

Figura 2 - Esquema de *Spooling*.



Fonte: SILBERSCHATZ, 2015.

Depois de 4 anos da primeira versão de lançamento do OS/360, em 1970 o projeto do sistema operacional UNIX foi iniciado. Esse sistema seria posteriormente a referência para a criação de uma família de sistemas operacionais, conhecidos como UNIX-Like (Baseado no UNIX).

Inspirado no projeto, Linus Torvalds criou um clone mais robusto, nomeado Linux, lançado em 1991 com foco em suprir demandas avançadas que outros sistemas não estavam dispostos a implementar.

A partir do lançamento a popularidade do Linux aumentou consideravelmente. Ser *open-source* (código aberto) facilitava a adesão de novos entusiastas do sistema e gerava o ímpeto de colaborar no projeto, que ganhou novas versões ao longo dos anos, chegando em 2020 a marca de 27.852.148 linhas de códigos (PHORONIX, 2020).

2.3 Processos

Conforme Tanenbaum (2016), o objetivo principal do computador é realizar tarefas de forma mais eficiente do que se fossem executadas por um humano. Cada tarefa em si, deve possuir a capacidade de ser requerida pelo usuário, executada, e retornar ao usuário o resultado quando necessário. Executando todo o procedimento sem que entre em conflito com outras tarefas presentes. Para que a ciência da computação pudesse projetar tal ideia, foi criado o conceito de Processo.

Como dito por Bovet (2005), o processo pode ser considerado uma instância de um programa em execução, que nasce com propósito de ser executado, pode gerar outro *child process* (processo filho), entrando na fila de processamento, e morrendo ao fim da execução. Este modelo simples e ultrapassado deu lugar ao suporte de aplicações *multithread*, que permitem ao programa, muitos fluxos de execução relativamente independentes e compartilhando parte das estruturas de dados da aplicação.

Ainda segundo Bovet (2005), para serem gerenciados, cada processo possui um *Process Descriptor* (Descritor de Processo), estrutura que contém todas as informações do processo, como a prioridade de execução, o endereço de espaço de memória que está e o Estado Atual do Processo (*Process State*). Com ciência desta estrutura, o sistema operacional consegue garantir maior flexibilidade, identificando contexto de execução separadamente.

O Linux identifica com um conjunto de IDs (Identifiers — Identificadores) próprios cada processo, nomeado de PID (*Process Identifier*, Identificador de Processo). Cada PID é reciclado por outro processo quando liberado, de forma circular, a fim de se manter o intervalo de números possíveis de identificação. Embora tenha 32.767 instâncias de PID de limite padrão, é possível a expansão manual até o número máximo de 4.194.303 PIDs em máquinas com arquitetura de 64-bit (BOVET, 2005).

Quando um processo precisa especificamente abrir um arquivo, ele não completa a atividade sozinho, sendo que há outras estruturas intermediárias. O sistema operacional repassa a função para o Sistema de Arquivos Virtual (*Virtual File System*, VFS), que usa a estrutura de CFM (*Common File Model*, Modelo de Arquivo Comum) para completar a tarefa, deixando pronto para o processo (BOVET, 2005).

Em casos críticos, o kernel pode usar *filesystem-dependent operations* (Operações dependentes do Sistema de Arquivos) para matar um processo existente e consequentemente abrir espaço na memória para alocação de uma nova página (Bovet, 2005).

2.4 Gerenciamento De Memória

No começo da computação, antes da criação de processos de gerenciamento de memória mais complexos, os dados simplesmente ficavam alinhados de forma sequencial, sem qualquer outra hierarquia mais complexa. Havia a ideia rudimentar de lista com campos de acesso independente pelo

programa em execução, que capturava o dado para um registrador para uso posterior (TANENBAUM, 2016).

A evolução de endereçamento de dados surgiu com OS/360, divisão por bloco com chaves de proteção próprias para cada bloco. Impedindo a sobrescrita de dados do sistema operacional ou outros dados importantes por programas sem permissão, sendo a implementação de um controle de acesso de memória.

Após essa abstração e o uso de um espaço de endereçamento de memória para cada processo, uma abertura para implementação de novas estruturas de dados surgiu, como mapas de bits e gerenciamento por lista encadeada. Mas com o aumento do uso de memória, teve que ser criado um espaço de memória separado para acesso sem congestionar todo o sistema de gerenciamento, dando vida ao conceito de memória virtual.

2.5 Entrada E Saída

Após a abstração dos acessos as portas de I/O (*Input/Output*, Entrada e Saída) do hardware em baixo nível usando linguagem *assembly*, os programadores do OS (*Operating System*, Sistema Operacional) tem a possibilidade de interpretar o fluxo de dados de forma original. Criando abstrações e estratégias que se adequem melhor ao restante da arquitetura do kernel.

O Linux implementa o I/O do sistema em uma estrutura similar aos sistemas de arquivos, porém direcionados para lidar com cada dispositivo de forma a abstrair o fluxo de dados, interpretando como gerenciamento de arquivos. Cada *device* (dispositivo) pode ser lido com o conjunto de ferramentas padrão do kernel sob a estrutura do VFS.

Esse sistema de padronização entra na categoria de *Pseudo* FS (*File System*, Sistema de Arquivos), facilitando o gerenciamento pelo OS, e sobrecarregando funções como **read()** (Leitura) e **write()** (Escrita) do sistema de arquivos implementados no kernel. Nesta estratégia não há a necessidade de que novas funções sejam criadas, que ocupariam espaço importante do sistema, simplificando a abstração.

As informações dos dispositivos do sistema necessárias para o I/O são expostas em formato hierárquico de objetos do kernel no SYSFS (*System File System*, Sistema de Arquivos do Sistema). Com as informações setadas e rotinas em execução, o VFS se comunica com os Device Drivers (Drivers de Dispositivo) para que o hardware responda ao processo solicitado e o fluxo de dados seja concluído.

CAPÍTULO III – SISTEMA DE ARQUIVOS

Este capítulo aborda a estrutura veiculada ao sistema de arquivos com foco no VFS, EXT4 (*Fourth Extended File System*, Quarto Sistema de Arquivos Estendido) e FUSE (*Filesystem in Userspace*, Sistema de Arquivos em Espaço de Usuário). Introduz a organização e a hierarquia da comunicação entre o kernel e o sistema de arquivos através do VFS. Apresenta a estrutura de blocos e sub-blocos do EXT4 e de seu sistema de *journal* JBD2. Apresenta de forma breve o sistema de arquivos em espaço de usuário, FUSE.

3.1 Introdução

Os Sistemas de Arquivos são componentes do sistema operacional que possuem grande variedade de tipos, sendo que diversos deles são usados como padrão no sistema mesmo que não possuam ponto de montagem para o usuário.

Estando abaixo do VFS na hierarquia do sistema, os sistemas de arquivos, têm um maior contato com os dados, sendo uma coleção de parâmetros de como gerenciar arquivos e diretórios em uma partição de disco ou espaço de memória reservado.

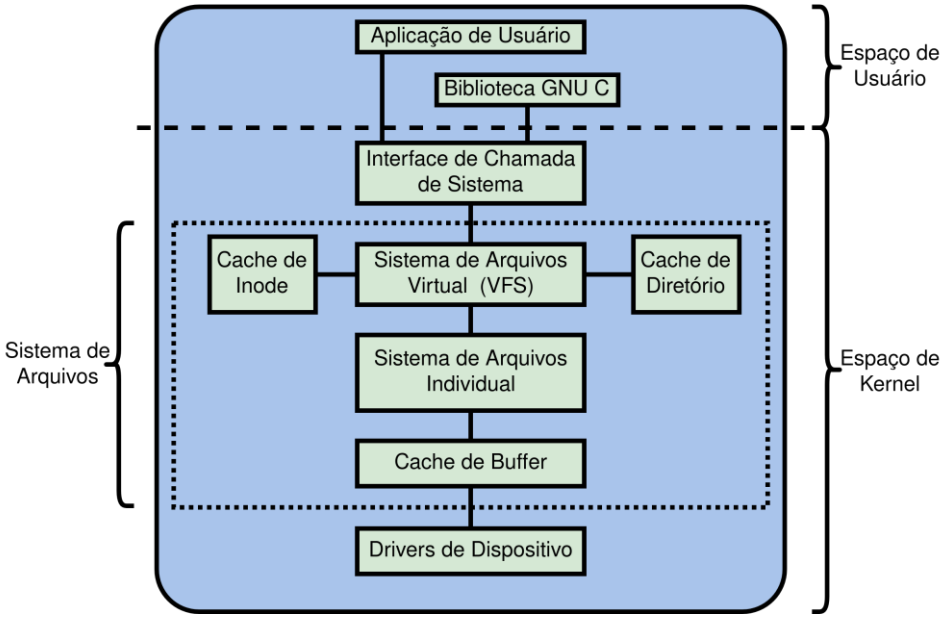
3.2 VFS

Assim como outras variantes do Unix, o Linux possui a capacidade de interagir e coexistir com outros discos e partições mesmo que estejam com diferentes tipos de sistemas de arquivo, isso é possível devido ao VFS (BOVET, 2005).

Chamado também de Sistema de Arquivos Virtual Interruptor (*Virtual Filesystem Switch*) pelo Linux Kernel Organization (2005), o VFS não se trata exatamente de sistema de arquivos, por não lidar diretamente com o gerenciamento de blocos de dados, sendo uma camada do kernel que implementa uma abstração.

Esse sistema tem o intuito de relacionar diferentes sistemas de arquivos de forma que não se tornem perceptíveis ao usuário durante a navegação no sistema de hierarquias de pastas e arquivos, abstraindo para uma uniformidade de acesso. Agindo como intermediário, como ilustrado na Figura 3, ele recebe todas as chamadas de sistemas de arquivos, padronizadas pelo POSIX (*Portable Operating System Interface*, Interface Portável para Sistemas Operacionais), as interpreta e generaliza em um acesso unificado.

Figura 3 – Contexto do Sistema de Arquivos Virtual no Linux.



Fonte: Adaptado de STALLINGS, 2015.

Este sistema pode ser interpretado como 2 segmentos conceituais principais, a parte superior que recebe a chamada do software e reinterpreta para cada sistema de arquivos que seja o alvo, e a parte inferior que recebe as chamadas dos diversos sistemas de arquivos e as afunila para a abstração final em um único sistema.

Cabe ao VFS manipular as chamadas de sistema (*system calls*) que são referentes ao sistema de arquivos, arquivos, diretórios e links simbólicos (BOVET, 2005), que estão listadas na Tabela 1.

Tabela 1 – Chamadas de Sistemas manipuladas pelo VFS.

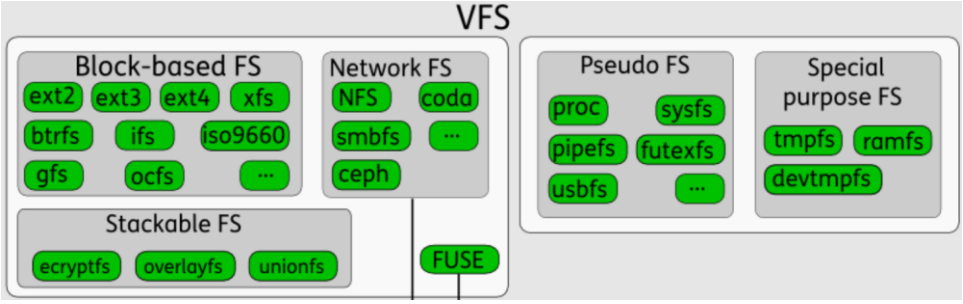
mount( ) umount( ) umount2( )	Monta/Desmonta sistema de arquivos
sysfs( )	Obtém informação do sistema de arquivos
statfs( ) fstatfs( ) statfs64() fstatfs64( ) ustat( )	Obtém estatística do sistema de arquivos
chroot( ) pivot_root( )	Altera diretório raiz
chdir( ) fchdir( ) getcwd( )	Manipula diretório atual
mkdir( ) rmdir( )	Cria e destrói diretórios
getdents( ) getdents64( ) readdir( ) link( ) unlink( ) rename( ) lookup_dcookie( )	Manipula entrada do diretório
readlink( ) symlink( )	Manipula soft links
chown( ) fchown( ) lchown( ) chown16( )	Modifica proprietário do arquivo

fchown16( ) lchown16( )	
chmod( ) fchmod( ) utime( )	Modifica atributos do arquivo
stat( ) fstat( ) lstat( ) access( ) oldstat( ) oldfstat( ) oldlstat( ) stat64( ) lstat64( ) fstat64( )	Lê status do arquivo
open( ) close( ) creat( ) umask( )	Abre, fecha, e cria arquivos
dup( ) dup2( ) fcntl( ) fcntl64( )	Modifica descritor do arquivo
select( ) poll( )	Espera por eventos em um conjunto de descritores de arquivo
truncate( ) ftruncate( ) truncate64( ) ftruncate64( )	Altera tamanho do arquivo
lseek( ) _llseek( )	Altera ponteiro do arquivo
read( ) write( ) readv( ) writev( ) sendfile( ) sendfile64( ) readahead( )	Realiza operações de I/O de arquivo
io_setup( ) io_submit( ) io_getevents( ) io_cancel( ) io_destroy( )	I/O assíncrono (permite solicitações de múltiplas leituras e escritas pendentes)
pread64( ) pwrite64( )	Procura arquivo e acessa-o
mmap( ) mmap2( ) munmap( ) madvise( ) mincore( ) remap_file_pages( )	Lida com mapeamento de memória de arquivo
fdatasync( ) fsync( ) sync( ) msync( )	Sincroniza dados do arquivo
flock( )	Manipula bloqueio do arquivo
setxattr( ) lsetxattr( ) fsetxattr( ) getxattr( ) lgetxattr( ) fgetxattr( ) listxattr( ) llistxattr( ) flistxattr( ) removexattr( ) lremovexattr( ) fremovexattr( )	Manipula atributos estendidos do arquivo

Fonte: Adaptado de BOVET, 2005.

O VFS integra diversos FSs, ilustrados na Figura 4, com finalidades específicas para suprir necessidades do usuário, como no caso do NFS (*Network File System*, Sistema de Arquivos de Rede). Este que permite que usuários e servidores compartilhem de um espaço de arquivos comum remotamente, com praticidade (TANENBAUM, 2016).

Figura 4 – Sistema de arquivos manipulados pelo VFS.



Fonte: THOMAS-KRENN (2021).

De acordo com Stallings (2015), o VFS é escrito em Linguagem C e esquematizado como orientado a objeto, sendo que os objetos contêm dados e ponteiros para as funções que manipulam os dados pelo FS. Os objetos implementados são em parte, um espelho das estruturas principais de um FS comum do sistema, como o EXT2 (*Second Extended File System*, Segundo Sistema de Arquivos Estendido).

O VFS tem como conceito principal de implementação a criação de um CFM, que representa todos os FSs suportados no OS. Neste conceito cada diretório é tido como um arquivo CFM diferente, que lista os arquivos e outros diretórios internos, correspondendo a hierarquia de arquivos real. Sendo uma reinterpretação orientada a objetos da hierarquia de diretórios dos sistemas.

Como listado por Bovet (2005), nessa estrutura de dados, os objetos podem ser do tipo: Objeto Super Bloco (*Superblock Object*), que guarda informações do FS montado; Objeto *Inode* (*Inode Object*), que guarda informações de um arquivo específico; Objeto Arquivo (*File Object*), que guarda informações entre arquivo e processo; Objeto de Entrada de Diretório (*Dentry Object*, *Object Directory Entry*), que guarda links de acesso de outros diretórios.

Cada objeto possui seus devidos atributos e métodos, sendo que o último pode ser alterado dinamicamente pelo kernel a fim de gerar um comportamento especializado e se adequar a algum FS específico.

3.3 Sistema De Arquivos EXT4

Do ponto de vista do usuário, uma das partes mais importantes do OS é o sistema de arquivos. O FS permite ao usuário criar e armazenar coleções de dados de forma organizada sem a preocupação de gerenciar endereços de memória diretamente (STALLINGS, 2015).

Os sistemas de arquivos, de forma geral, alocam usando uma série de blocos, cada um com uma função diferente, que tem sua organização agrupada e replicada, formando uma partição do disco.

Assim, cada partição pode ser configurada em um FS específico que posteriormente pode ser montada pelo VFS para acesso, sem que entre em conflito com outro sistema.

Como documentado por Linux Kernel Organization (2005), o EXT4 tem todos os campos escritos no disco em *little-endian*. Disponibilizando, redundância interna em árvore, pré-alocação de arquivos persistentes e suporte a grandes arquivos.

3.3.1 *Inodes*

O armazenamento de um arquivo não é feito de forma contínua, sendo fragmentado para aproveitar todos os espaços possíveis, dividindo um arquivo em vários nós (*nodes*). Para identificar cada bloco do arquivo, após serem separados, é usada a estrutura de dados *inode* (*index node*, índice de nós).

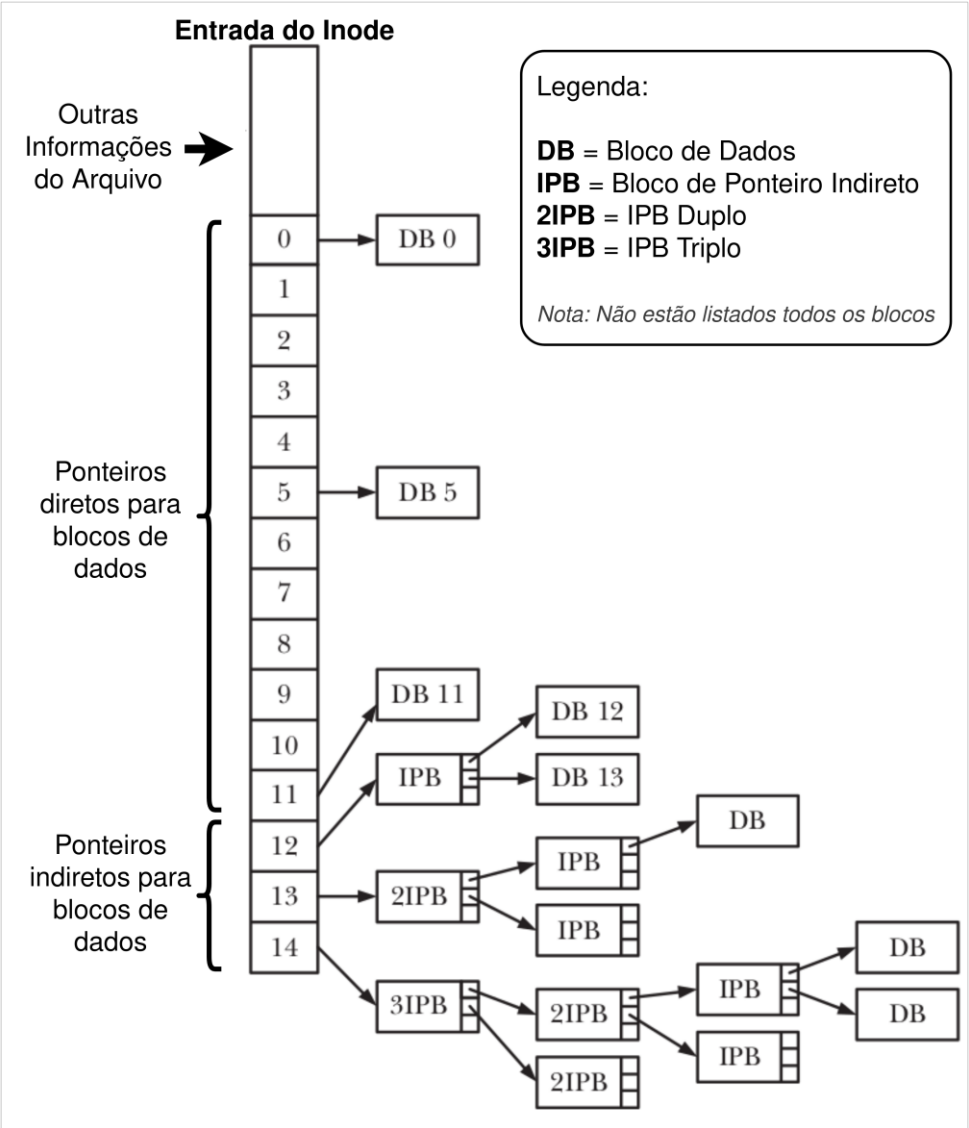
No *inode* são gravadas todas as informações de *metadados* do arquivo, salientando o tipo de arquivo, ID do proprietário, tamanho em bytes e o número de blocos alocados para o arquivo.

Como diagramado por Kerrisk (2010), e ilustrado na Figura 5, cada registro de *inode* possui 15 ponteiros, os primeiros 12 apontando para blocos do arquivo dividido, enquanto o restante são ponteiros de ponteiros indiretos, duplos e triplos, para uso em caso de arquivos maiores.

A maioria das variáveis dessa estrutura de dados são correspondentes com a especificação do POSIX com o intuito de facilitar a implementação junto ao VFS. Relacionando *inode object* do VFS e *inode* do sistema de arquivos. Como resultado, poucas alterações no inode ocorreram entre as versões do sistema EXT2 ao EXT4.

Conforme documentado em Linux Kernel Organization (2005), nas versões de sistemas EXT2 e EXT3 (*Third Extended File System*, Terceiro Sistema de Arquivos Estendido), o tamanho da estrutura de *inode* é fixada em 128 bytes, e de valor flexível no EXT4. Possibilitando alocações maiores e que condizem com o avanço do armazenamento computacional.

Figura 5 – Estrutura de bloco de arquivos para um arquivo no sistema EXT2.

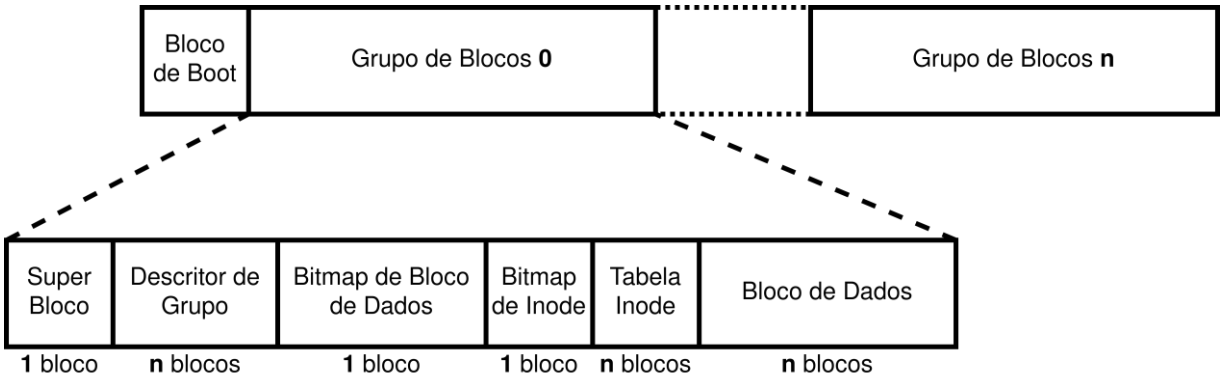


Fonte: Adaptado de KERRISK (2010).

3.3.2 Blocos

Em uma partição comum, o FS divide o espaço de memória em 2 tipos de blocos, como ilustrado na Figura 6, o Bloco de Boot (*Boot Block*) e inúmeros Grupos de Blocos (*Block Groups*) subsequentes. Cada um com seus metadados específicos em variáveis estáticas.

Figura 6 – *Layout* de partição de sistema de arquivos EXT2.



Fonte: Adaptado de BOVET, 2005.

O primeiro bloco da partição é preenchido com o Bloco de Boot que é usado para armazenar informações de boot do sistema operacional. Sendo ignorado pelo sistema de arquivos na maioria das vezes. O espaço restante da partição é preenchido por Grupo de Blocos, sob a estratégia de dividir a indexação e consequentemente diminuir erros de sobrescrita de dados.

Cada Grupo de Blocos tem um conjunto completo de sub-blocos, com funções individuais. Como listado por Bovet (2005), em ordem de preenchimento, estão:

Super Bloco (*Super Block*) que indica parte substancial das especificações de armazenamento e do sistema de arquivo;

Descritor de Grupo (*Group Descriptor*) que armazena as características do grupo em que está inserido;

Bitmap de Bloco, ou Bitmap de Bloco de Dados (*Data Block Bitmap*) que mapeia quais blocos de dados estão vazios ou preenchidos;

Bitmap de *Inode* (*Inode Bitmap*) que mapeia quais *inodes* estão vazios ou preenchidos;

Tabela *Inode* (*Inode Table*) que armazena o registro de cada arquivo ou diretório do bloco;

Bloco de Dados (*Data Blocks*) que armazena os dados internos dos arquivos e diretórios.

3.3.3 Super Bloco

O super bloco é estruturado em campos com diferentes tamanhos, guardando informações sobre as configurações do sistema de arquivos de forma geral. Nos atributos contidos, se destacam o número total de *inodes*, número e tamanho de blocos reservados, número de blocos e *inodes* por grupo, contador de blocos e *inodes* livres, e sistema operacional em que o sistema de arquivos foi criado.
Esses campos de metadados variam entre tipos sem sinal (*unsigned*) e com sinal (*signed*), sendo alocados na estrutura **ext4_super_block**.

Os super blocos salvam cópias redundantes de super blocos de outros grupos, facilitando reparo quando algum deles é corrompido, porém diminuindo o espaço para dados, já que são estruturas estáticas.

Quando um FS é montado usando o comando **mkfs**, a *flag* **sparse_super** pode ser setada para alterar o padrão de redundância de super blocos. Dessa forma, as cópias de redundância acontecerão somente no grupo 0 e nos múltiplos de 3, 5, ou 7 (LINUX KERNEL ORGANIZATION, 2005).

3.3.4 Descritor De Grupo

Cada descritor de grupo, assim como o super bloco, também guarda uma cópia da Tabela de Descritor do Grupo (*Group Descriptors Table*), tabela que registra com redundância os descritores de grupo de outros grupos, a menos que a *flag* **sparse_super** esteja setada.0
Em conjunto com o super bloco, os dois são os únicos com local fixo registrado, sendo que os outros blocos podem variar dinamicamente de tamanho após a criação do sistema.
Nele são armazenados o endereço do bitmap de bloco, bitmap de *inode* e tabela *inode* do grupo. Também se destacam a contagem de blocos e *inodes* livres, e a contagem de diretórios.

3.3.5 Tabela *Inode*

Conforme Bovet (2005), tabela *inode* é responsável por conter um número de *inodes* necessários para aquele segmento, com a quantidade de *inodes* predefinida durante a montagem do sistema de arquivo e armazenada na variável **bg_inode_table** do descritor de grupo (BOVET, 2005). Sendo um *array* linear da estrutura de dados *Inode* com todos os *inodes* dinâmicos (LINUX KERNEL ORGANIZATION, 2005).

3.3.6 Bitmaps

Bitmaps são matrizes de bits em que cada bit representa, no bitmap de bloco, o uso de um bloco de dados, e no bitmap de *inode*, a entrada de *inode* no grupo. Em que o valor 0 especifica o espaço livre e o valor 1 especifica que está em uso (BOVET, 2005).

3.3.7 Journal

Como explicado por Tanenbaum (2016), *journal* é um diário criado para o sistema de arquivos, que registra todas as ações que serão realizadas antes que o sistema as faça, agindo como proteção para os metadados caso ocorra inconsistências após uma falha do sistema.

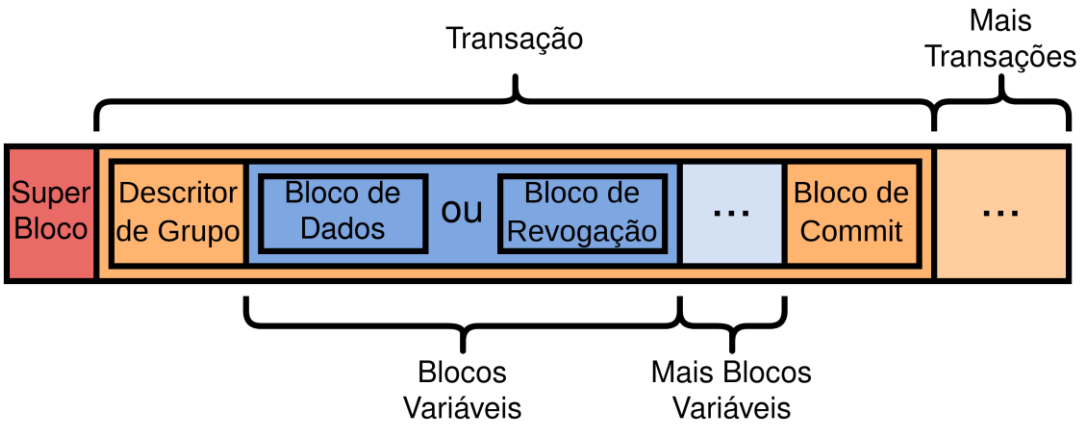
O *journaling* (método de utilizar o *journal* no sistema de arquivos) elimina a necessidade de que todo o sistema seja checado para que inconsistências no sistema de arquivos sejam reparadas e os metadados dos blocos sejam revalidados. Sendo que o método de checagem completo possa levar horas para ser executado (Kerrisk, 2010).

Introduzido no EXT3, e passado ao EXT4, o *journal* por predefinição somente registra os metadados, ignorando os blocos de dados. Porém se a opção **data=journal** estiver setada durante a montagem, os metadados e os arquivos passarão pelo registro, diminuindo a velocidade, mas melhorando a segurança.

O EXT4 usa a versão de Dispositivo de *Journaling* em Bloco 2 (*Journaling Block Device 2*, JBD2) de *journal*, que tem todos os campos preenchidos em *big-endian*, indo de encontro ao EXT4. A estrutura do JBD2 é ilustrada na Figura 7, dividida em blocos, com alguns homônimos aos do FS EXT4. Tipificados como Super Bloco (*super block*), Descritor de Bloco (*descriptor block*), Bloco de Dados (*data block*), Bloco de Revogação (*revocation block*) e Bloco de *Commit* (*commit block*).

O sistema de arquivos também pode ser criado com um *journal* externo. Então o *journal* deixa de usar os *inodes* reservados do sistema de arquivo e cria um device separado.

Figura 7 – Layout de blocos do sistema de *journal* JBD2.



Fonte: Elaborado pelo autor, 2021.

3.3.7.1 Super Bloco

O super bloco do *journal* possui menor complexidade que o super bloco contido no EXT4. Os principais dados mantidos são, o tamanho do *journal* e a localização dos registros de transações. O super bloco é registrado como a estrutura **journal_superblock_s**, que possui 1024 bytes de comprimento (LINUX KERNEL ORGANIZATION, 2005).

3.3.7.2 Descritor De Bloco

Conforme documentado em Linux Kernel Organization (2005), O descritor de bloco do *journal* possui um *array* **journal_block_tag_s**, estrutura que contém as *tags* necessárias para descrever a localização final dos blocos de dados que estão registrados no *journal*. Esta estrutura de *array* e *tags* pode variar dependendo da flag setada durante a configuração do *journal*.

3.3.7.3 Bloco De Dados

Os blocos de dados registrados usando *journal* são escritos diretamente após o descritor de bloco do *journal*. Exceto quando os primeiros quatro bytes do bloco correspondem ao "*magic number*" do JBD2. No caso os bytes serão setados com o valor 0 e a *flag* **escaped** será setada no descritor de bloco do *journal* (LINUX KERNEL ORGANIZATION, 2005). Essa checagem é realizada para distinguir os metadados do *journal*, dos blocos de dados (ARPACI-DUSSEAU, 2005).

3.3.7.4 Bloco De Revogação

O bloco de revogação é usado para prevenir a repetição de blocos contidos em uma transação anterior. Usado para marcar blocos que passaram pelo registro, mas não estão mais nele. Isso acontece se um bloco de metadados é liberado e realocado como bloco de dados de arquivo. No caso, uma repetição depois do bloco de arquivo ser escrito em disco causaria uma inconsistência (LINUX KERNEL ORGANIZATION, 2005).

3.3.7.5 Bloco De *Commit*

O bloco de *commit* é um marco que indica que a transação foi completamente registrada no *journal*. No momento que esse bloco é escrito, os dados registrados com a transação podem ser gravados nos locais definidos do disco (LINUX KERNEL ORGANIZATION, 2005).

3.3.7.6 *Checkpoint*

O checkpoint em um *journal* é usado para assegurar que todas transações e buffers foram submetidas no disco. É usado internamente durante situações críticas como atualizações do sistema de arquivos, recuperação de *journal* e redimensionamento do sistema de arquivos.

O checkpoint também pode ser chamado em espaço de usuário usando o comando **ioctl** e a *flag* **EXT4_IOC_CHECKPOINT** (LINUX KERNEL ORGANIZATION, 2005).

3.4 Sistema De Arquivos FUSE

Conforme documentado por Linux Kernel Organization (2005), FUSE é um framework que inclui o módulo de kernel (*fuse.ko*), a biblioteca de usuário (*libfuse.**) e utilitário de montagem (*fusermount*).

Ainda segundo Linux Kernel Organization (2005), este framework é a interface base para a criação de um Sistema de Arquivos do Espaço de Usuário (*Userspace Filesystem*), sistema de arquivos em que os dados e metadados são fornecidos por processos comuns do usuário, possibilitando montagem e gerenciamento seguro sem a necessidade de acesso privilegiado.

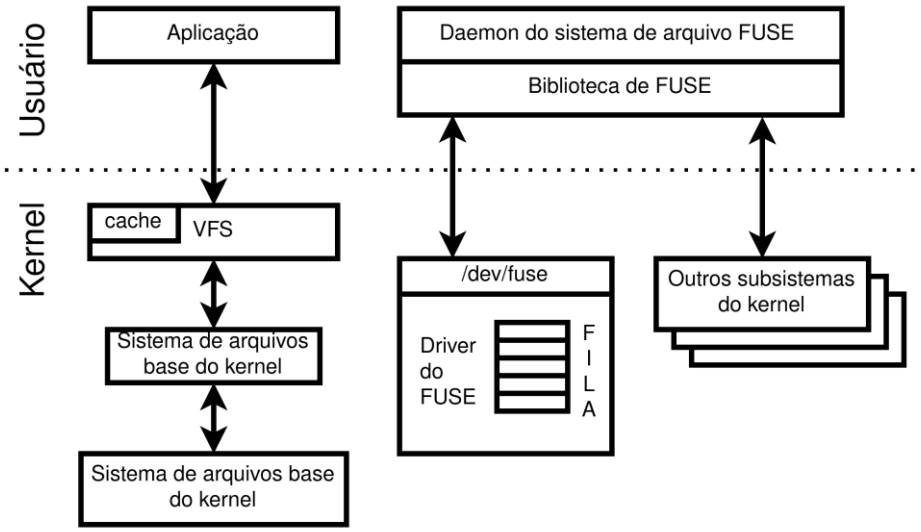
Com o framework é possível o desenvolvimento de um sistema de arquivos com características divergentes dos outros sistemas já compatíveis com o VFS sem que seja necessária a implementação nativa adicional ao kernel.

O sistema FUSE, como ilustrado na Figura 8, se divide em duas partes, o sistema de arquivos "fuse" implementado como módulo do kernel e que se comunica com o VFS de forma semelhante aos demais FSs, e um *daemon* em nível de usuário que age como intermediário, denominado *User-Space Daemon* (*Daemon* em Espaço de Usuário) (VANGOOR, 2017).

A parte do kernel fica contida em **/dev/fuse**, quando carregado registra um driver de sistema de arquivos FUSE com VFS. Este driver de FUSE (*Fuse Driver*) atua como um proxy para vários sistemas de arquivos específicos implementados por diferentes daemons de nível de usuário (VANGOOR, 2017).

O *daemon* responsável possui a denominação básica de *Filesystem Daemon* (*Daemon* de Sistema de Arquivo) na documentação oficial ou *User-Space Daemon* (Daemon de Espaço de Usuário) e *FUSE File-System Daemon* (*Daemon* de Sistema de Arquivo FUSE) em outras documentações e artigos. Fato é que para denominar de forma geral os processos que fornecem os dados e metadados para o módulo FUSE e consequentemente o FUSE Driver é usado o termo "*Filesystem Daemon*".

Figura 8 – Arquitetura do FUSE em alto nível.



Fonte: Adaptado de VANGOOR, 2017.

A montagem do sistema de arquivos FUSE pode ser feita com o comando **mount**, repassando os parâmetros para montar um FS FUSE de forma usual ou **fuseblk** que se baseia em montar um device em bloco que pode ser nomeado.

Em sequência, a passagem de parâmetros se destaca o **fd** que indica o descritor de arquivo (*file descriptor*), o **default_permissions** que indica o modo de permissões, o **allow_other** que altera o acesso, e o **blksize** que seta o tamanho do bloco.

O descritor de arquivo atribuído como parâmetro é usado para a comunicação entre o FS do usuário e o kernel, no caso, contido no FUSE device (**/dev/fuse**).

O **default_permissions** habilita a checagem de permissões de acesso do FS montado, se baseando no modo individual de cada arquivo, sendo contrário ao padrão de permissão do FUSE, que repassa ao usuário a implementação de um sistema de política de acesso. Esse parâmetro pode funcionar em conjunto com o **allow_other** que habilita o acesso para outros usuários além do que está montando o sistema.

O último salientado, **blksize**, somente é permitido no modo **fuseblk**, alterando o valor de 512 de tamanho padrão.

Sendo que a ferramenta **mount** requer acesso root para a maioria dos usos, faz se necessário a utilização do **fusermount**, um programa para montar e desmontar especificamente sistemas de arquivo FUSE.

De acordo com Linux Kernel Organization (2005), o *fusermount* possui 3 princípios para proporcionar uma ferramenta de montagem segura:

- A- O usuário que execute o montador não deve conseguir privilégios elevados a partir do sistema de arquivos montado;
- B- O usuário que execute o montador não deve obter acesso ilegítimo às informações dos processos de outros usuários;
- C- O usuário que execute o montador não deve ser capaz de induzir comportamento indesejado nos processos de outros usuários.

Após a montagem do FS, o *daemon* do sistema de arquivos e o módulo de kernel criam uma conexão denominada Conexão de Sistema de Arquivos (*Filesystem Connection*), que permanece mesmo na ocorrência de desanexação de device (*device detaching*) por *lazy unmounting*. Encerrando quando o sistema for desmontado ou o *daemon* for morto (LINUX KERNEL ORGANIZATION, 2005).

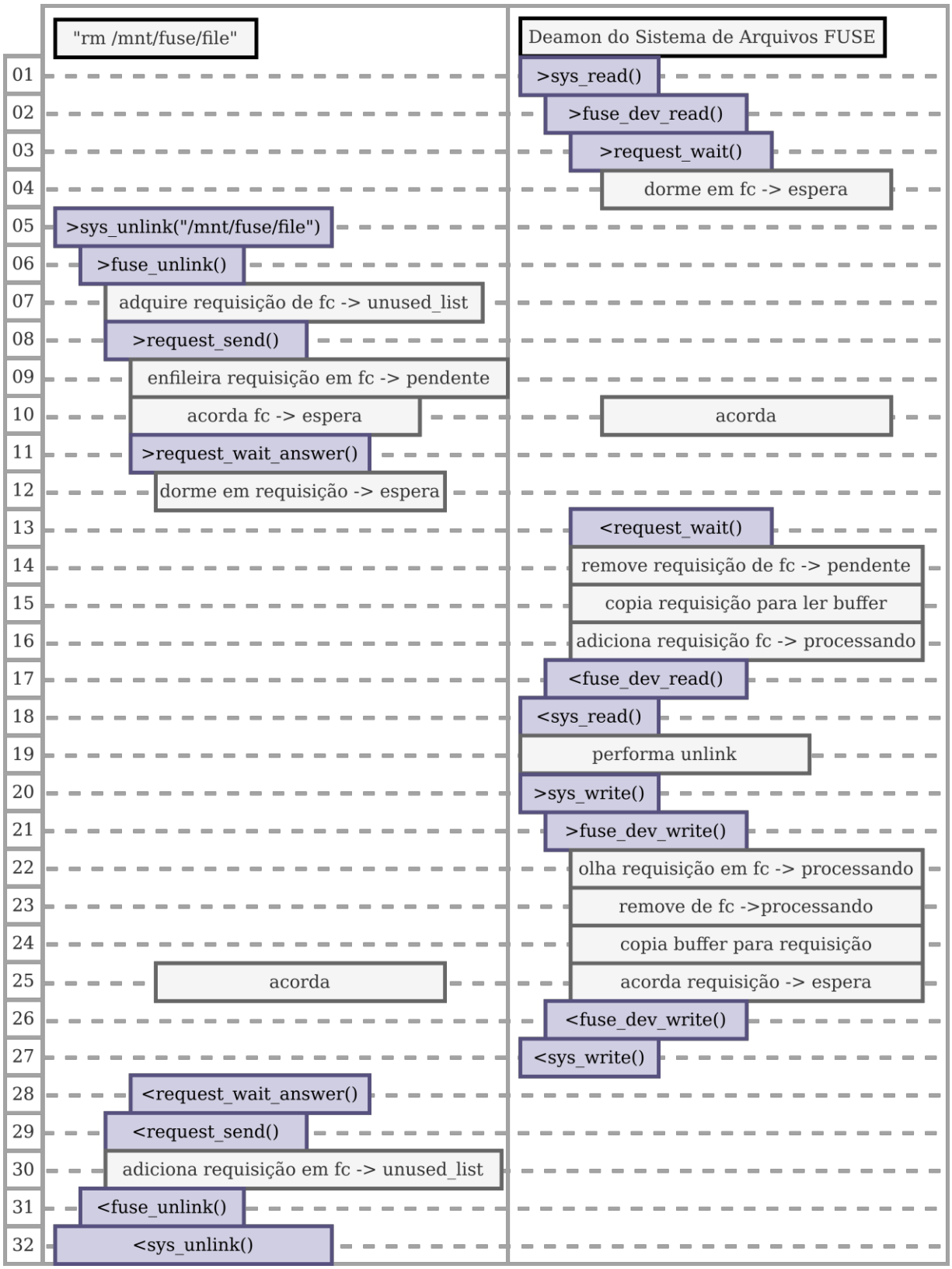
Com a conexão de sistema de arquivos estabelecida, todas as requisições do usuário que se relacione com o FS montado serão repassadas para o VFS, direcionadas ao driver de FUSE e consequentemente colocadas em estado de espera e enfileiradas para serem executadas pelo *daemon*.

O *daemon* processa as requisições em ordem, se comunicando com os Drivers de Dispositivos ou outros módulos do kernel que forem necessários, como no exemplo de operação performada pelo FUSE, ilustrado na Figura 9.

A resposta do processamento segue o caminho inverso, passando pelo driver de FUSE que marca como concluída e acorda a requisição antes em modo de espera, repassando a informação até a aplicação a nível de usuário.

Muitas requisições que somente leem arquivos que já estão em cache de página podem ser processadas sem que sejam repassadas ao driver de FUSE (VANGOOR, 2017).

Figura 9 – Exemplo de operação de *unlink* performada pelo FUSE.



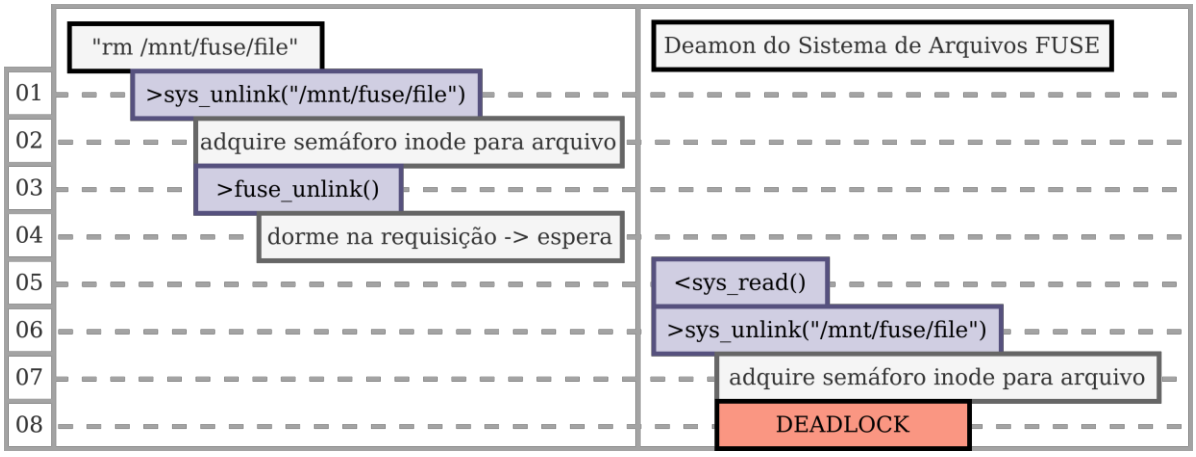
Fonte: Adaptado de LINUX KERNEL ORGANIZATION, 2005.

Para se prevenir de possíveis deadlocks, como o ilustrado na Figura 10, devido a comunicação dentre os sistemas, é implementada a situação de abortagem do sistema. Se protegendo de deadlocks acidentais e maliciosos, como também de perda de conexão de rede e quebra na implementação do sistema de arquivos no espaço de usuário.

Os métodos em ordem crescente de eficiência para executar a abortagem são, matar o *daemon* do sistema de arquivos, matar todos os usuários do sistema de arquivos, forçar a desmontagem do sistema de arquivos e abortar o sistema de arquivos pelo controlador de sistemas de arquivos do FUSE.

Além da abortagem, é implementado o processo de interrupção de requisições, usado para erros menores de processos que possam travar o enfileiramento do driver de FUSE (LINUX KERNEL ORGANIZATION, 2005).

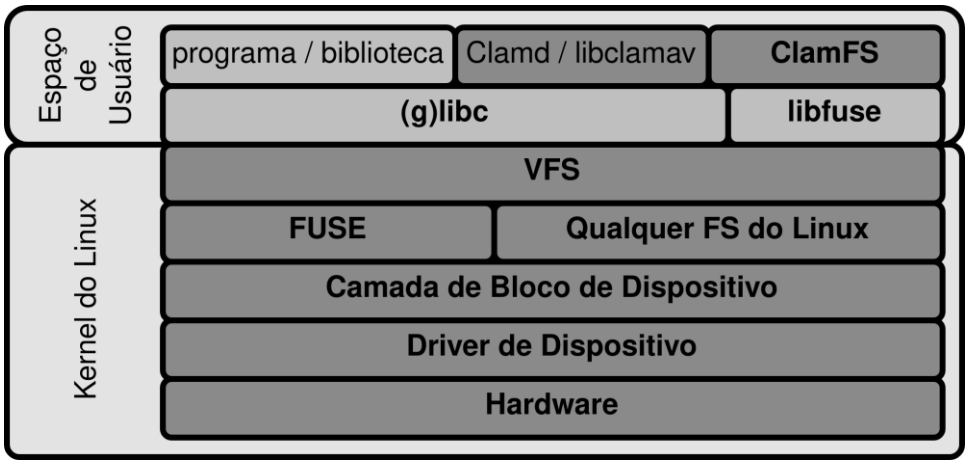
Figura 10 – Exemplo de deadlock simples no sistema de arquivos FUSE.



Fonte: Adaptado de LINUX KERNEL ORGANIZATION, 2005.

O FUSE abre possibilidades de criação para inúmeros FS com propostas inovadoras, como o SSHFS, um FS de rede para se conectar a servidores SSH, ou o *ClamFS*, com arquitetura ilustrada na Figura 11, que provê varredura de antivírus nos arquivos.

Figura 11 – Arquitetura do FUSE em alto nível usado no *ClamFS*.



Fonte: Adaptado de CLAMFS PROJECT PAGE, 2020.

CAPÍTULO IV – ESTEGANOGRAFIA

Este capítulo aborda o básico da esteganografia, como seu histórico e conceitos principais, apresentando a estrutura geral de funcionamento.

4.1 Introdução

A esteganografia é a arte de esconder informação de formas que previnam a detecção de mensagens escondidas (JOHNSON, 1998). Embora, seja frequentemente ligada a criptografia por ambas pertencerem ao estudo da criptologia, a esteganografia se difere por contemplar somente a ação de ocultar a informação. Enquanto a criptografia foca nas ações que tornam a informação ininteligível.

Os métodos mais comuns de esteganografia são os que focam em imagens ou sinais em geral, que dissolvem a informação a ser escondida em um arquivo alternativo, que será mantido ou compartilhado em rede sem que levante suspeita. Como mostrado na Tabela 2, a esteganografia não proporciona certeza quanto a confiabilidade e integridade, abrindo espaço para muitas discussões em relação à efetivação de seu uso.

Tabela 2 – Comparação de Técnicas de Comunicação Secretas.

Técnicas de Comunicação Secretas	Confidencialidade	Integridade	Irremovibilidade
Encriptação	Sim	Não	Sim

Assinaturas Digitais	Não	Sim	Não
Esteganografia	Sim / Não	Sim / Não	Sim

Fonte: Adaptado de BHADRA, 2014.

4.2 História

Os primeiros registros de esteganografia foram feitos pelo historiador grego Heródoto que em suas histórias registrou duas técnicas (BHADRA, 2014). A primeira técnica se tratava de escrever uma mensagem no couro cabeludo raspado, para que quando o cabelo voltasse a crescer, a mensagem fosse camuflada. A segunda técnica relatada usava tabuletas cobertas por cera para escrita cotidiana da época, Figura 12, em que a mensagem era gravada diretamente na madeira e posteriormente coberta por cera para criar a ocultação.

Figura 12 – Tábua coberta de cera usada na Grécia Antiga para a escrita.

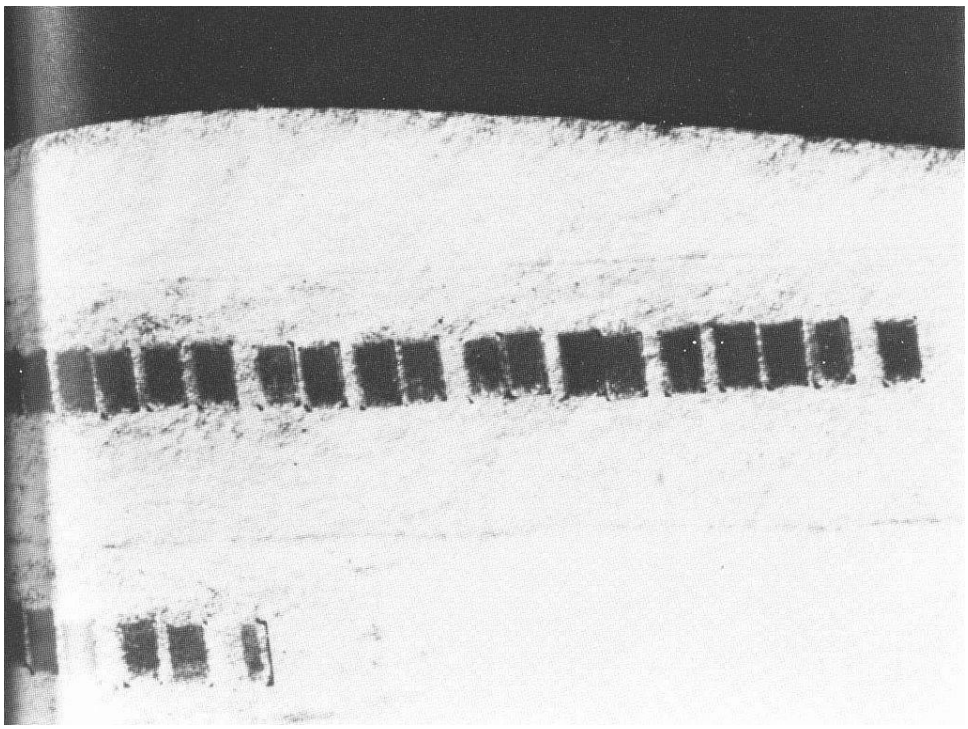
Fonte: WAGONER, 2012.

Conforme Johnson (1998), ao longo da história, pessoas esconderam informações e, portanto, uma infinidade métodos foram criados. Foram desenvolvidos desde tinta invisível, a marcação de posição de linhas e palavras em um texto, e chegando às práticas mais sofisticadas como micropontos, ilustrados na Figura 13.

Grande parte das inovações da esteganografia ocorreram durante conflitos, como a primeira e segunda guerra mundial, momento em que censures agiam interceptando e alterando mensagens enviadas, caso fossem informações sensíveis. Nestes casos a esteganografia era essencial para manter a integridade da mensagem e não levantar suspeita (PETITCOLAS, 1999).

Com o avanço da tecnologia, a esteganografia alcançou a computação, abrindo possibilidades para novos métodos, como o comum Bit Menos Significativo (*Least Significant Bit*, LSB), e novos propósitos, como proteção de marcas de copyright através de técnicas de marcas d'água (*watermarking techniques*) (BHADRA, 2014).

Figura 13 – Micropontos gravados dentro da etiqueta de um envelope enviado por agentes alemães no México para Lisboa.



Fonte: MOWRY, 2011.

4.3 Definições Básicas

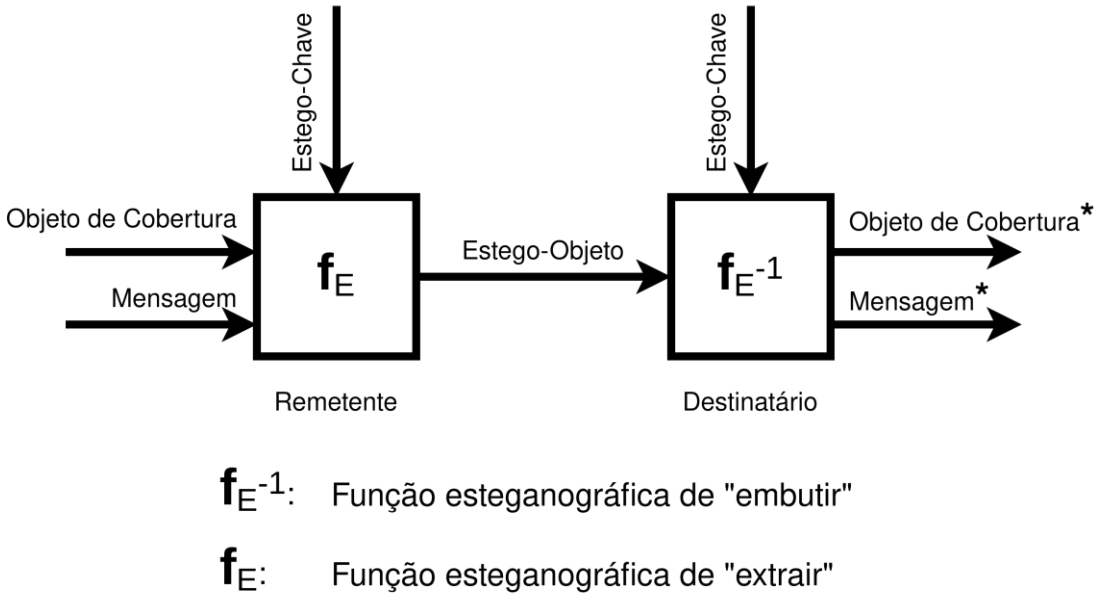
A esteganografia clássica se atenta ao ato de embutir (*embed*) a mensagem secreta (*secret message*) em uma mensagem ou objeto de cobertura (*cover-object*). O embutimento usado é comumente parametrizado como chave (*key*). Uma vez que a mensagem secreta tenha sido incorporada no objeto de cobertura, este resultado recebe o nome de estego-objeto (*stego-object*) (PETITCOLAS, 1998).

Embora a tradução mais fiel seja “embutir”, o sentido literal indica algo ligeiramente diferente do conceito de esteganografia computacional. A mensagem secreta não será colocada “dentro” do objeto de cobertura, e sim, incorporada a fim de criar uma mensagem que é a junção das anteriores. Criando um estego-objeto, estego-texto, estego-imagem, etc. Denominação que melhor se adequar ao resultado do embutimento.

Petitcolas (1999), a informação já dentro do estego-objeto é denominada de dado ou mensagem embutida (*embedded*) e o processo para embutir e resgatá-la é tido como estego-chave (*stego-key*).

Assim, para realizar o processo de esteganografia é preciso a mensagem secreta, a mensagem de cobertura e a estego-chave, como ilustrado na Figura 14. Para reverter o processo é preciso a estego-chave, o estego-objeto e, em alguns casos, o objeto de cobertura.

Figura 14 – Versão gráfica de sistema esteganográfico.

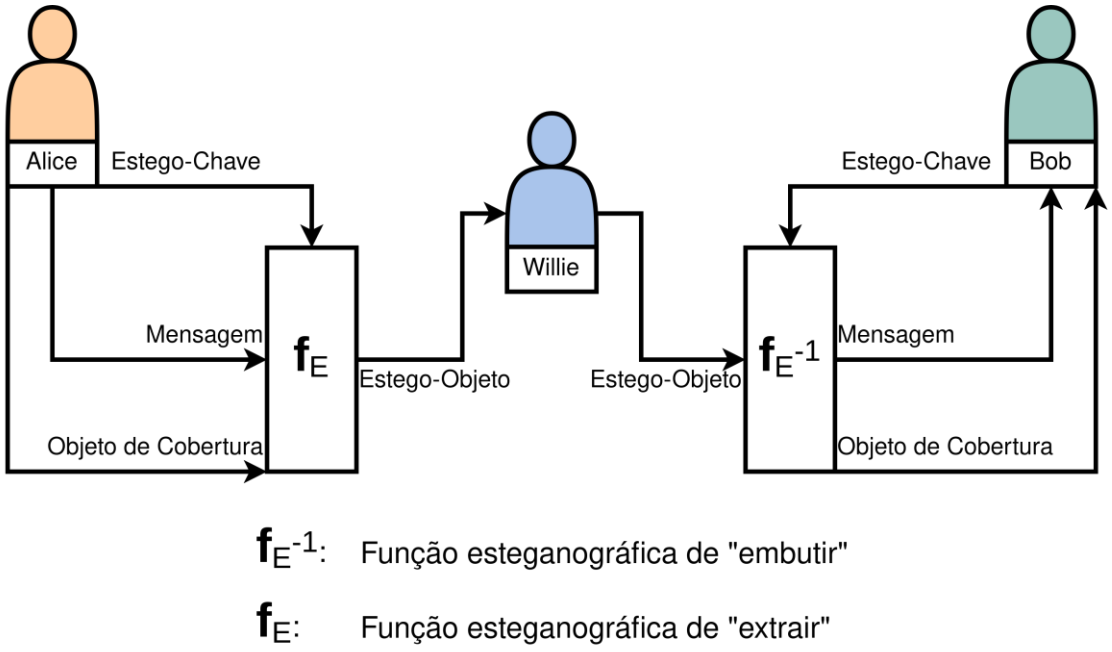


Fonte: Adaptado de BHADRA, 2014.

É essencial que mesmo que o sistema usado seja descoberto, este seja impossível de ser executado sem uma chave adequada. Neste caso, são favoráveis o uso de ataques contra esteganografia, esteganálise, que usam diferentes técnicas para detectar e extrair as mensagens secretas dos possíveis estego-objetos.

Sistemas de Esteganografia de chave pública são teorizados para a criação de um ecossistema de chaves que não necessitem exclusivamente de chaves secretas, mesmo percalço já ultrapassado pela criptografia. Porém, o próprio sentido de público entra em conflito com a ideia principal da esteganografia. Para que a estrutura de uma chave pública seja desenvolvida é preciso relevar que a ciência de um processo esteganográfico possa quebrar a comunicação, algo fundamental para o, ilustrado na Figura 15, problema dos prisioneiros de Simmon (*Simmon's prisoner's problem*) proposto em 1983 (AHN, 2004).

Figura 15 – Diagrama do Problema dos Prisioneiros de Simmon.



Fonte: Elaborado pelo autor, 2021.

CAPÍTULO V – SISTEMA DE ARQUIVO ESTEGANOGRÁFICO

Este capítulo apresenta e especula sobre métodos de esteganografia à nível de sistema de arquivos. Lista e comenta implementações vigentes. Como também, supõe outros possíveis métodos.

5.1 Introdução

Unir esteganografia com o sistema de arquivos denota um novo nível de segurança além da criptografia já amplamente usada. Sistemas de arquivos criptografados indicam aos atacantes que uma chave possivelmente está de posse do usuário, abrindo precedentes para ataques, como *phishing*, a fim de extrair mais informações.

Como dito por SHAMIR (1998), um sistema de arquivos esteganográfico de sucesso pode proporcionar ao usuário maior proteção contra coerção, de forma que a possam negar de forma plausível a existência de diretórios inteiros, mesmo contra um oponente que tenha acesso direto ao disco rígido.

5.2 Método XOR

A primeira ideia, proposta por Shamir (1998), era de usar uma senha P com o tamanho de um número pré-definido de bits, k. O sistema inicialmente cria um conjunto de arquivos que serão preenchidos com dados aleatórios, servindo como objetos de cobertura, de forma que todos tenham o mesmo tamanho. O arquivo secreto, é então embutido em um subconjunto de arquivos dentre todos os objetos de cobertura, de forma que a operação XOR linear de todos os arquivos usados retorne o arquivo embutido, como ilustrado na Figura 16.

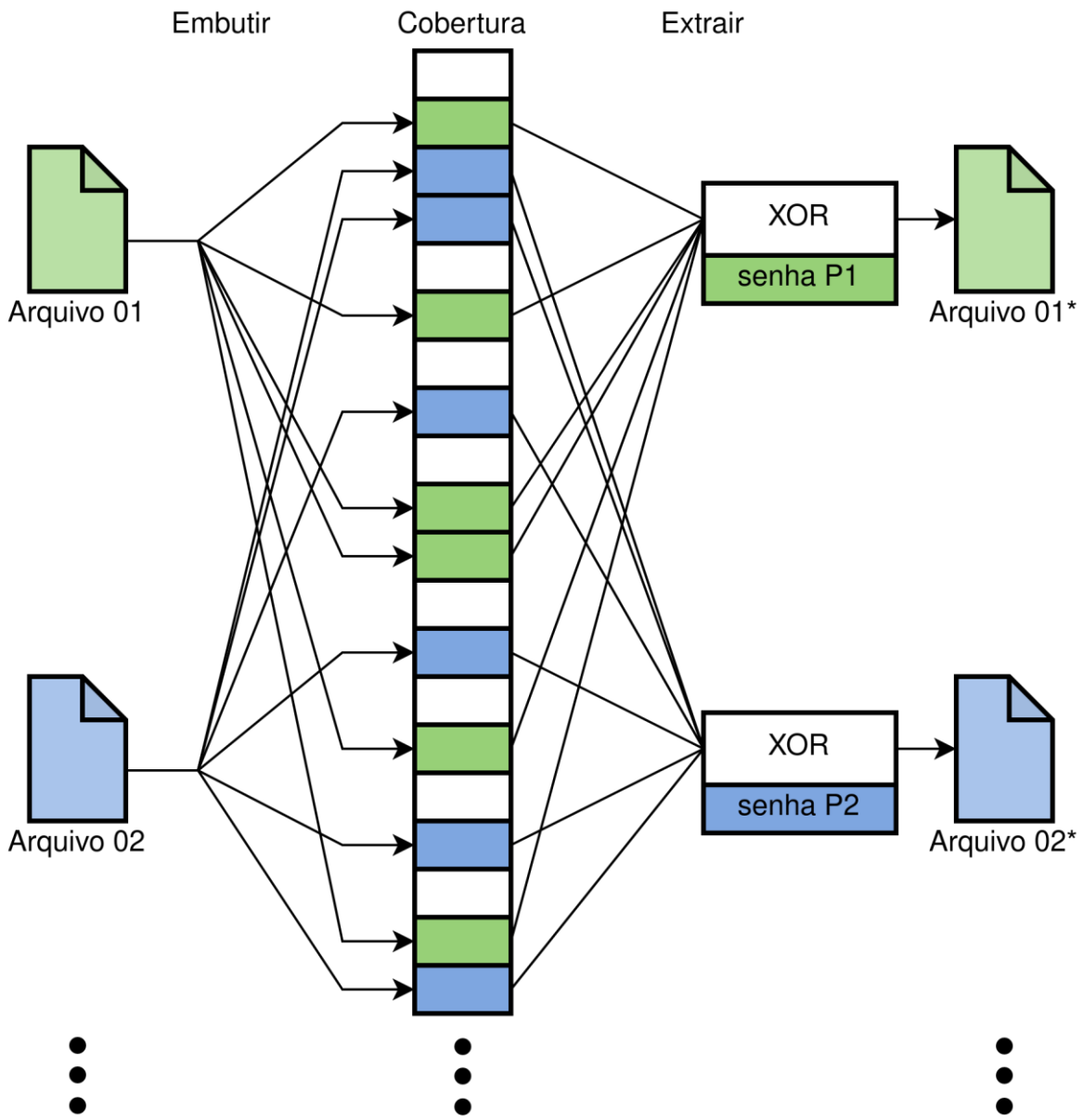
Conceitualmente, papel de estego-objeto é dividido dentre os arquivos do subconjunto, que serão escolhidos pela senha P. O usuário gerencia a senha P de cada arquivo de modo que os próximos subconjuntos escolhidos não tenham arquivos em intersecção com os subconjuntos anteriores, a fim de que nenhuma parte do estego-objeto seja reescrita indevidamente. A listagem de senhas geradas seria posteriormente criptografada por uma senha única do usuário.

Vários fatores devem ser desempenhados para que este sistema seja computacionalmente seguro. A quantidade de arquivos que compõem os objetos de cobertura deve ser suficientemente grande para que os arquivos secretos sejam embutidos sem que haja necessidade de colisão entre os arquivos escolhidos, além de criar uma margem de erro para que seja inviável computacionalmente para o atacante usar a operação XOR em teste de força bruta. Outro percalço é a aleatoriedade dos dados abertos dos objetos de cobertura que, para um atacante, seria interpretado como dados criptografados, não o dissuadindo de que são arquivos comuns.

Mesmo que toda a teoria do sistema leve em consideração que o atacante é completamente ignorante sobre o conteúdo dos arquivos do objeto de cobertura, cabe também ao atacante não saber nenhuma parte da mensagem, o que contribui para a inviabilidade de implementação (MCDONALD, 1999).

Considerar um sistema que lide com uma leitura e escrita de arquivos tão dispendiosa não é totalmente fora do comum, levando em consideração que o sistema de *journaling* ainda não estava totalmente consolidado nos sistemas de arquivos. Embora o uso de porta lógica XOR como parte central da ideia agilizasse o processo, com arquivos pequenos, verificar todo o sistema de arquivos em caso de inconsistências ainda era computacionalmente aceitável.

Figura 16 – Sistema de arquivos esteganográfico com método XOR.



Fonte: Elaborado pelo autor, 2021.

5.3 *StegFS*

Concebido por Shamir (1998) e posteriormente repensado por McDonald (1999), o *StegFS* (*Steganographic File System*, Sistema de Arquivos Esteganográfico) tem como princípio de funcionamento, alocar arquivos em partes não usadas do sistema de arquivos padrão mantendo uma tabela própria de alocação.

Como descrito por McDonald (1999), a tabela de alocação funciona como o bitmap de bloco do FS, porém ao invés manter um bit para cada bloco, contém uma chave de 128 bits como entrada. Está

sempre presente no sistema mesmo se não usado e é mantido criptografado para que, mesmo que sua presença acuse que o *StegFS* está presente no sistema, maiores informações não podem ser extraídas.

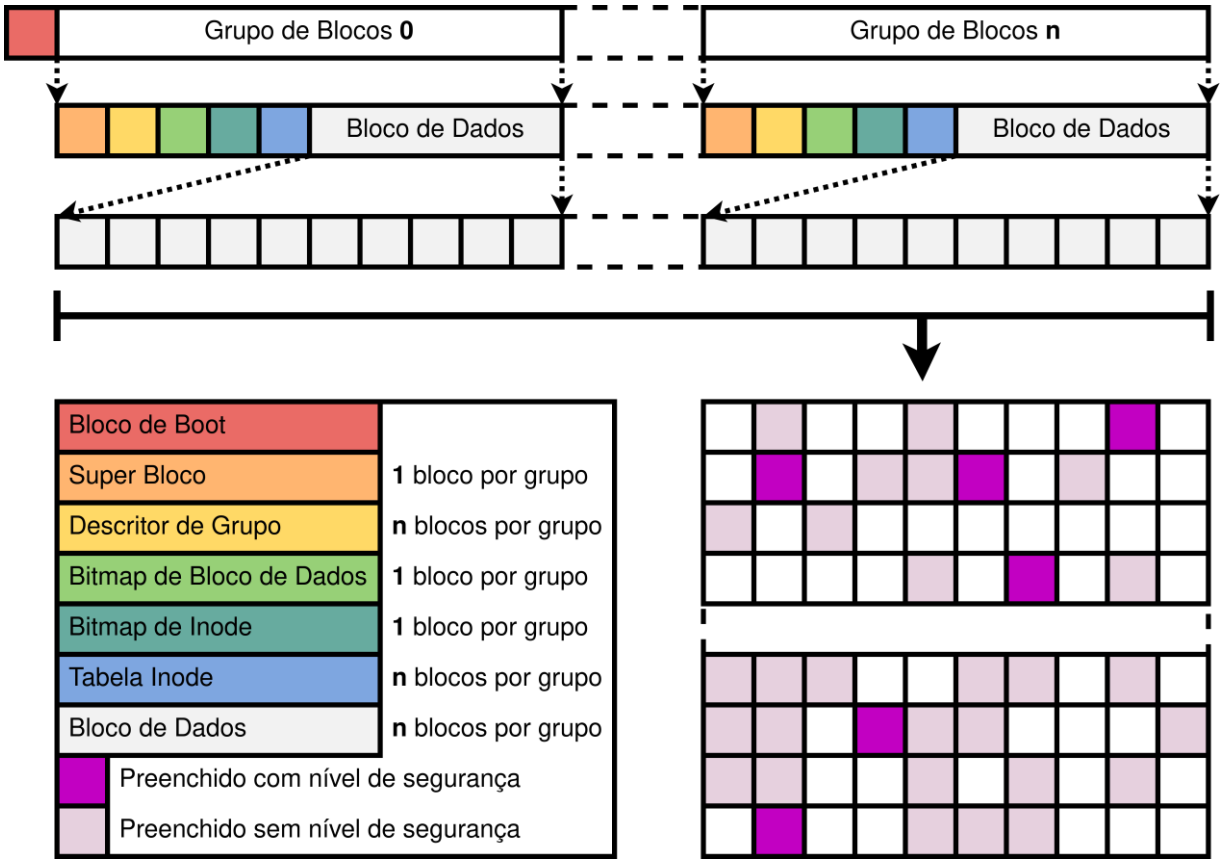
Agindo como um sistema de upgrade, o *StegFS* é montado sobre uma partição já montada com EXT2, o processo preencherá os blocos vazios do com dados aleatórios e criando a tabela de alocação denominada de tabela de bloco (*Block Table*) como um arquivo separado e não-embutido (MCDONALD, 1999). A partir do ponto simples de montagem, o *StegFS* age de forma mímica ao EXT2, como ilustrado na Figura 17 manipulando arquivos não embutidos no sistema, com exceção da exclusão de arquivos que substitui imediatamente os blocos de dados com bytes aleatórios.

A manipulação de dados embutidos requer o uso de um comando separado que modifica o nível de segurança do *StegFS* e consequentemente permite o modo de esteganografia do sistema. Com a ferramenta **stegfsopen** com a configuração de um maior nível de segurança N (1 a 15), o sistema libera a pasta "stegfs" na raiz da partição para acesso que consequentemente contém acesso um mesmo número N de diretórios para esteganografia. Isso cria uma hierarquia de acesso, em que 0 seria o nível mímico ao EXT2.

Sendo que o arquivo de tabela de blocos somente é verificado em níveis de segurança maiores que zero, ao modificar blocos de dados dentro do sistema, o *StegFS* lerá exclusivamente os blocos de metadados internos do EXT4 caso nenhum nível de segurança esteja setado. Ignorando o registro de blocos de dados embutidos, o sistema pode ocasionalmente selecionar um bloco preenchido, criando sobrescritas. Configurando o *StegFS* como um FS Com Perdas *(Lossy*) (MCDONALD, 1999).

Embora o *StegFS* consiga embutir dados com segurança de acesso, o sistema não contempla completamente os princípios da esteganografia, sendo que tanto o driver do sistema quanto o arquivo de tabela de bloco ficam visíveis. Outro percalço é a perda por sobrescrita que induz ao usuário registrar cópias do mesmo arquivo em diferentes níveis de segurança, a fim de diminuir a probabilidade que seja corrompido. Se aproveita da estrutura do EXT2 que sendo prévio ao *journaling*, possui menor capacidade de rastreio dos blocos computados, melhorando a chance de que dados embutidos passem despercebidos.

Figura 17 – Layout de blocos do sistema *StegFS*.



Fonte: Elaborado pelo autor, 2021.

5.4 Esteganografia Em FUSE

Após o avanço do sistema de arquivos, ficou mais oneroso o desenvolvimento de sistemas de arquivos separadamente, pois as permissões privilegiadas necessárias abriam precedentes perigosos de uso. Além de que o avanço do *journal* impede boa parte da esteganografia de agir de forma mais sofisticada no sistema. O uso do FUSE abriu novas portas, o desenvolvedor poderia implementar um FS sem a necessidade de maiores permissões do sistema, agindo com maior discrição, parte fundamental da esteganografia.

A ideia mais implementada seria manter o padrão de acesso de um FS comum assim como o *StegFS*, porém, a estrutura não visa embutir arquivos dentro dos blocos não usados do sistema. Seria repassado para o FS em FUSE, os algoritmos e métodos de esteganografia comumente utilizados para a criação de estego-objetos, de forma que substituísse a necessidade da existência dos softwares esteganográficos instalados no sistema operacional. Ao montar o FS FUSE sobre outro sistema de arquivo, ele executaria os algoritmos em todos os arquivos existentes e caso o algoritmo fosse bem-sucedido, o arquivo resultante seria retornado no próprio diretório do estego-objeto processado.

Método que se assemelha ao um algoritmo de extração em modo recursivo, verificando todos os arquivos dentro dos diretórios da partição, verificando os estego-objetos presentes e extraindo para o usuário, lidando com vários arquivos em somente uma montagem.

Embora exista o custo alto de verificar todos os arquivos do FS, já que não existe um arquivo de registro como a tabela de bloco do *StegFS*, o FUSE se aproveita das chamadas padronizadas POSIX do VFS que acelera o processo, com prioridade acima uma aplicação comum a nível de espaço de usuário.

Fato que, embora o uso da esteganografia seja usado em conjunto do FS FUSE, esse método não pode ser considerado à nível de sistema de arquivos como o ideal teorizado, já que manipula arquivos separadamente como um algoritmo comum, sem se aproveitar da arquitetura real do FS.

Outros projetos pontuais de implementação visam usar a mesma estratégia do StegFS, porém ainda sem solução para a perda de dados por sobrescrita, critério que só piorou com o sistema de *journaling* no EXT4.

CAPÍTULO VI – CONSIDERAÇÕES FINAIS

Neste capítulo são apresentadas as considerações finais deste Trabalho de Conclusão de Curso, assim como dificuldades encontradas durante o desenvolvimento e sugestões para trabalhos futuros.

6.1 Conclusão

Este Trabalho de Conclusão de Curso teve como objetivos apresentar um estudo sobre sistema de arquivos com foco no EXT4 e FUSE, como também suas relações com o VFS e o sistema operacional Linux, seguindo para uma leitura e especulação de Esteganografia pode ser implementada à nível de sistema de arquivos.

Inicialmente foi criada uma pequena introdução com a história e as funcionalidades principais do sistema operacional Linux, citando interações pontuais com o sistema de arquivos e VFS.

Foi abordado, o VFS e sua comunicação intermediária entre o kernel e o FS, como também suas peculiaridades e estrutura. Descreveu-se de forma profunda o sistema de arquivos EXT4, seus blocos de estrutura, assim como o seu sistema de *journaling* e sua estrutura interna e interação. Foi explicada a forma de ação do FUSE e sua relação com o kernel em comparação com EXT4.

Apresentou-se o básico da esteganografia, como contextualização histórica da criação e conceitos-chave necessários para o trabalho. Foi realizada uma análise dos sistemas de arquivos esteganográficos mais relevantes, citando suas estratégias, funcionamento, qualidades e defeitos.

Levando em consideração os dados levantados, o objetivo do trabalho foi parcialmente alcançado, pouca documentação e implementação de esteganografia em nível de sistema de arquivos relevante foi encontrada, levando a um enriquecimento do trabalho ou possibilidade vigente de especulação sobre métodos pouco comentados. Sendo que os estudos realizados nos capítulos prévios tenham sido realizados de forma satisfatória.

6.2 Dificuldades Encontradas

Para o desenvolvimento deste trabalho, foram necessários maiores levantamentos bibliográficos do que os já exigidos para cursar uma disciplina regular. Sendo vital o uso de um sistema mais robusto de estudo e organização.

Ao longo do desenvolvimento, faz-se difícil o uso de material bibliográfico pouco atualizado, direcionando-se a análise de código fonte e documentação de desenvolvedores. O assunto é atualmente pouco desenvolvido e cabe ao longo da pesquisa a verificação de conceitos já ultrapassados.

Foram enfrentadas diversas situações de luto devida à crise sanitária ocorrida durante o período de desenvolvimento, que consequentemente prejudicou o avanço potencial do trabalho. A divergência dos paradigmas comuns de ensino, mesmo que contornados, geraram atrasos no cronograma.

6.3 Trabalhos Futuros

Sugere-se como trabalhos futuros:

- Estudo e atualização das estratégias levantadas para melhor adequação com a tecnologia de sistemas de arquivos atuais;
- Estudo, implementação de um sistema de arquivos esteganográfico com alguma das estratégias levantadas;
- Estudo e implementação de um FUSE FS focado na esteganografia.

REFERÊNCIAS BIBLIOGRÁFICAS

AHN, Luis von; Hopper, Nicholas J. Public-Key Steganography. **Lecture Notes in Computer Science**, Berlin, v. 3027, p. 323-341, 2004.

ARPACI-DUSSEAU, Andrea; ARPACI-DUSSEAU, Remzi; PRABHAKARAN, Vijayan. Analysis and Evolution of Journaling File Systems. **Proceedings of the 2005 USENIX Annual Technical Conference**, Anaheim, CA, USA, p. 105-120, abr. 2005.

BHADRA, Jayati; BOJAMMA, A.M.; PRASAD, C.N.; NACHAPPA, M.N. An Insight to Steganography. **IJISET - International Journal of Innovative Science, Engineering & Technology**, v. 1, n. 10, p. 29-42, dez. 2014. ISSN 2348-7968.

BOVET, Daniel P.; CESATI, Marco. **Understanding the Linux Kernel**. 3ª Edição, Sebastopol, CA: Editora "O'Reilly", 2005.

CLAMFS PROJECT PAGE. **SourceForge**, 2020. Disponível em: <http://clamfs.sourceforge.net/>. Acesso em: 20 mai. 2021.

CORBET, Jonathan; KROAH-HARTMAN, Greg; RUBINI, Alessandro. **Linux Device Drivers**. 3. ed. Sebastopol, CA: Editora "O'Reilly", 2005.

DOOLER, Matthew. **A Diagnostic Tool for FUSE Modules**. Orientador: Dr. Graham Kirby. Senior Honours Project - University of St Andrews, St. Andrews, Scotland, 2014.

JOHNSON, Neil F.; JAJODIA, Sushil. Exploring Steganography: Seeing the Unseen. **Computer**, Washington, DC, USA, v. 31, n. 2, p. 26-34, fev. 1998. ISSN 0018-9162.

KERRISK, Michael. **The Linux Programming Interface: a Linux and UNIX System Programming Handbook**. San Francisco, CA: No Starch Press, 2010.

LAUREANO, Marcos Aurelio; OLSEN, Diogo Roberto. **Sistemas Operacionais**. 1. ed. Curitiba: Editora do Livro Técnico, 2010.

LINUX KERNEL ORGANIZATION. **Filesystems in the Linux kernel**. Disponível em: <https://www.kernel.org/doc/html/latest/filesystems/index.html>. Acesso em: 20 mai. 2021.

MCDONALD, Andrew D.; KUHN, Markus G. StegFS: A Steganographic File System for Linux. **Lecture Notes in Computer Science**. USA, v. 1768, p. 462-477, abr. 1999.

MCDONALD, Nicolas G. **Past, Present and Future Methods of Cryptography and Data Encryption**. Departamento de Engenharia Elétrica e de Computação, Universidade de Utah, Estados Unidos, 2010.

MOWRY, David P. **Cryptologic Aspects of German Intelligence Activities in South America during World War II**. MD, EUA: Center for Cryptologic History National Security Agency, 2011.

PETITCOLAS, Fabien A.P.; ANDERSON, Ross J. On The Limits of Steganography. **IEEE Journal of Selected Areas in Communications**, v. 16, p. 474-481, maio 1998. ISSN 0733-8716.

PETITCOLAS, Fabien A.P. et al. Information Hiding—A Survey. **In Proceedings of IEEE. Special issue on Protection on multimedia content**, v. 87, p. 1062-1078, jul. 1999.

PHORONIX. **GitStats - linux**. 01 jan. 2020. Disponível em: <https://www.phoronix.com/misc/linux-eoy2019/index.html>. Acesso em: 14 abr. 2021.

SHAMIR, Adi; NEEDHAM, Roger; ANDERSON, Ross. The Steganographic File System. **Lecture Notes in Computer Science**, USA, v. 1525, p. 73-82, abr. 1998.

SILBERSCHATZ, Abraham; GALVIN, Peter Baer; GAGNE, Greg. **Fundamentos de Sistemas Operacionais**. 9ª ed. Rio de Janeiro: LTC, 2015.

STALLINGS, William. **Operating Systems: Internals and Design Principles**. 8.ed. Upper Saddle River: Pearson Prentice Hall. 2015.

TANENBAUM, Andrew S. et al. **Sistemas Operacionais Modernos**. 4. ed. São Paulo: Pearson Education do Brasil, 2016.

THOMAS-KRENN. **Linux Storage Stack Diagram**. Disponível em: <https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram>. Acesso em: 20 mai. 2021.

VANGOOR, Bharath et al. To FUSE or Not to FUSE: Performance of User-Space File Systems. **Proceedings of the 15th USENIX Conference on File and Storage Technologies**, Santa Clara, CA, USA, p. 1-49, fev. 2017.

WAGONER, Brady. Culture in Constructive Remembering. **The Oxford Handbook of Culture and Psychology**, jan. 2012.

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
GABINETE DO REITOR

Av. Universitária, 1069 ● Setor Universitário
Caixa Postal 86 ● CEP 74605-010
Goiânia ● Goiás ● Brasil
Fone: (62) 3946.1000
www.pucgoias.edu.br ● reitoria@pucgoias.edu.br

RESOLUÇÃO n° 038/2020 – CEPE

ANEXO I

APÊNDICE ao TCC

Termo de autorização de publicação de produção acadêmica

O(A) estudante **Gabriel Oliveira Laureano** do Curso de **Ciência da Computação**, matrícula **2019.2.0028.0031-1**, telefone: _____ e-mail **gabcomp.gov@gmail.com**, na qualidade de titular dos direitos autorais, em consonância com a Lei nº 9.610/98 (Lei dos Direitos do autor), autoriza a Pontifícia Universidade Católica de Goiás (PUC Goiás) a disponibilizar o Trabalho de Conclusão de Curso intitulado **ESTEGANOGRAFIA EM SISTEMA DE ARQUIVOS DO SISTEMA OPERACIONAL LINUX**, gratuitamente, sem ressarcimento dos direitos autorais, por 5 (cinco) anos, conforme permissões do documento, em meio eletrônico, na rede mundial de computadores, no formato especificado (Texto (PDF); Imagem (GIF ou JPEG); Som (WAVE, MPEG, AIFF, SND); Vídeo (MPEG, MWV, AVI, QT); outros, específicos da área; para fins de leitura e/ou impressão pela internet, a título de divulgação da produção científica gerada nos cursos de graduação da PUC Goiás.

Goiânia, **13** de **dezembro** de **2021**.

Assinatura do(s) autor(es): _____

Nome completo do autor: **Gabriel Oliveira Laureano.**

Assinatura do professor-orientador: _____

Nome completo do professor-orientador: **Olegário Correa da Silva Neto, MSc.**