Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Remoção Segura de Arquivos em EXT3: técnicas para evitar a recuperação de dados.

Thiago Rodrigues Cunha

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Ms. João José Costa Gondim

Brasília
2014

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Homero Luiz Piccolo

Banca examinadora composta por:

Prof. Ms. João José Costa Gondim (Orientador) — CIC/UnB
Prof. Professor I — CIC/UnB
Prof. Professor II — CIC/UnB

CIP — Catalogação Internacional na Publicação

Cunha, Thiago Rodrigues.

Remoção Segura de Arquivos em EXT3: técnicas para evitar a recuperação de dados. / Thiago Rodrigues Cunha. Brasília : UnB, 2014.
243 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2014.

1. Forense Digital, 2. Exclusão de arquivos, 3. Antiforense Digital,
4. Formatação, 5. Limpeza de Disco, 6. Sistemas de arquivos

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Remoção Segura de Arquivos em EXT3: técnicas para evitar a recuperação de dados.

Thiago Rodrigues Cunha

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Ms. João José Costa Gondim (Orientador)
CIC/UnB

Prof. Professor I
CIC/UnB

Prof. Professor II
CIC/UnB

Prof. Dr. Homero Luiz Piccolo
Coordenador do Bacharelado em Ciência da Computação

Brasília, 18 de agosto de 2014

Dedicatória

Dedico esta monografia a toda minha família pela compreensão e pela fé demonstrada. Aos meus amigos, que tanto me apoiaram e me deram força. Aos professores por estarem sempre dispostos a compartilhar os seus conhecimentos. Aos orientadores pela paciência e pelo empenho em fazer o melhor. Enfim a todos que de alguma forma tornaram este sonho menos arduo de se tornar real.

Agradecimentos

Agradeço a Deus pela oportunidade de poder contribuir com os meus conhecimentos a toda a sociedade. A minha família pelo incentivo e colaboração. A toda a Universidade de Brasilia e seus membros que favoreceram essa conquista intelectual. Ao meu orientador, João Gondim, pela disposição e pelos conhecimentos compartilhados. Aos meus colegas pelas palavras amigas, pelos insentivos ao trabalho e por estarem comigo nesta caminhada. E principalmente a minha amada pela compreensão e pela disposição em querer sempre o melhor para mim.

Resumo

Este trabalho apresenta um estudo de baixo nível com análise de estruturas de dados do sistema de arquivos EXT3 com o intuito de propor e avaliar estratégias para evitar a recuperação de arquivos. Desenvolvemos uma ferramenta capaz de atuar nas estruturas de dados desse sistema e oferecemos uma análise sobre a remoção segura um arquivo baseado no funcionamento dos *softwares* de recuperação de dados.

Palavras-chave: Forense Digital, Exclusão de arquivos, Antiforense Digital, Formatação, Limpeza de Disco, Sistemas de arquivos

Abstract

This paper presents a study of low-level analysis of structures data from EXT3 file systems in order to propose and evaluate strategies to prevent files data recovery. We developed a tool able to work on EXT3 data structures and provide an analysis on wiping files based on the operation of data recovery software .

Keywords: Digital Forensics, Data Wipe, Secure Wipe, Format, File system, File deletion

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Os computadores tornaram-se importantes para boa parte das atividades da vida moderna. Desde empresas ligadas a área financeira, como bancos e cooperativas, às relacionadas a saúde, como hospitais, a preocupação com a segurança de dados torna-se essencial tanto para manter a confiança dos usuários como para sustentar as empresas no mercado. Não basta apenas os sistemas protegerem os seus dados contra acessos não autorizados, mas é necessário também que se preocupem com a perda e o roubo de informações importantes.

Muito se tem estudado a respeito de técnicas de segurança para evitar a violação de dados. Com o uso crescente de sistema sofisticados de criptografia, atacantes com a intenção de obter acesso a dados sensíveis são forçados a empregar seus esforços em novos métodos afim de obter melhores resultados, conforme Gutmann [7] menciona em seu artigo. Desde a década de 80, muitos são os estudos relacionados à recuperação de dados em discos magnéticos. Ao longo desse tempo, bons resultados foram conseguidos. Existem casos relacionados a discos danificados que tiveram 99% dos dados recuperados. De acordo com Dan Kaminsky, conseguir destruir 100% de dados em um disco, ou seja, torná-los irrecuperáveis, é praticamente impossível [17].

Se por um lado temos pessoas trabalhando para realizar a recuperação de dados computacionais, existem outras lidando com informações extremamente críticas e que jamais podem parar em mãos erradas. Essas são as chamadas informações sensíveis ou classificadas.

De acordo com o [16], a sensibilidade é uma característica dada à informação ou ao conhecimento que pode resultar em uma perda de vantagem ou nível de segurança se esta for revelada para terceiros. Logo, a perda, o mau uso, a alteração, ou o acesso não autorizado a ela, pode afetar a privacidade e o bem-estar de um indivíduo. Segredos comerciais de uma empresa ou até mesmo a informações que afetariam a segurança interna e as relações exteriores de uma nação, são exemplos de informações sensíveis.

Joan Feldman, presidente e fundador da *Computer Forensics* em Seattle compara o computador a um gravador continuamente em operação [6]. Ele registra cada palavra que alguém digita, e muitas vezes mantém inúmeras cópias desses dados visando a proteção dos usuários. Basta realizar uma busca dentro de algumas pastas do Windows para notar diversos arquivos temporários. Em cada um deles é possível obter informações de sessões capazes de recriar uma imagem do que estava acontecendo no computador em um determinado momento. Aplicações como o Word automaticamente salvam versões

temporárias de um documento em intervalos regulares com o único intuito de ajudar os usuários caso ocorra algum problema como uma queda súbita de energia. O que a maioria delas não imaginam é que todas essas versões temporárias de seus documentos podem mais tarde ser recriadas por outra pessoa. Tao fato pode não parecer tão importante quando se trata apenas de lembretes diários, mas pode se tornar traumático caso o documento restaurado contenha uma lista com dados pessoais e salários dos principais executivos de uma empresa. Infelizmente apenas encontrar todos os arquivos necessários e jogá-los na lixeira também não resolverá o problema.

Os sistemas de computadores tornam-se mais sofisticados e suas aplicações mais diversificadas a cada dia. A necessidade de proteger a integridade dos dados neles manipulados cresceu consideravelmente. Nesse trabalho criamos uma ferramenta (chamada FRS) capaz de excluir arquivos e sobrescrever dados das estruturas de um sistema EXT3. Analisamos, também, a capacidade de recuperação de alguns softwares após seu uso visando contribuir com a segurança de dados armazenados em computadores. Dessa forma, ele é um estudo do sistema de arquivos EXT3 visando a proteção de dados de arquivos que foram removidos.

1.1 Objetivo

O trabalho tem por objetivo contribuir com a segurança de dados sensíveis de usuários de um sistema operacional. Ele é um estudo a respeito da remoção segura de dados e visa propor técnicas para inviabilizar ou, ao menos, dificultar a recuperação de dados em sistemas de arquivos.

1.2 Estrutura do Trabalho

O presente trabalho foi dividido da seguinte forma.

O Capítulo 2 retrata de maneira superficial os sistemas de arquivos usados em computadores. Aborda mídias de armazenamento, armazenamento de informações através da eletricidade, finalizando com a estrutura criada para organizar dados e gerar informações no sistema operacional.

O Capítulo 3 discorre sobre o sistema de arquivo EXT3 especificando cada uma de suas partes e como o conjunto funciona ao realizar operações de alocação e de exclusão de arquivos.

No Capítulo 4, o leitor encontrará algumas técnicas utilizadas na recuperação de dados com ênfase nas estruturas do sistema de arquivos EXT3.

O Capítulo 5 falará a respeito da ferramenta (FRS) desenvolvida, como parte desse trabalho, para tentar automatizar a destruição de dados em disco. Avaliamos, assim, seu impacto na recuperação de arquivos. Consta também detalhes do experimento realizado e os seus resultados.

Por fim, o capítulo 6 apresenta as considerações finais sobre o estudo, direcionamento para trabalhos futuros e, na forma de Apêndice, o código usado da (FRS).

Capítulo 2

Sistemas de Arquivos

Computadores podem guardar informações em diversos dispositivos, como discos magnéticos, fitas, discos óticos ou de estados sólidos (SSD). Sistemas e softwares, de maneira geral, não existiriam caso não fosse possível armazenar ou restaurar dados. No quesito computadores, as informações aparecem na forma de unidades chamadas **arquivos**.

A parte do sistema operacional que mantém os arquivos organizados, realiza operações sobre eles e garante sua segurança e sua integridade é denominada **sistema de arquivos**.

A expressão sistema de arquivos nos sugere dois significados. O primeiro é a parte do sistema operacional responsável pela árvore de diretórios e sua coleção de arquivos. O outro significado é referente ao tipo de sistema de arquivos, ou seja, como os dados do computador são organizados no disco ou em uma de suas partições. Cada tipo de sistema de arquivos possuem suas próprias regras para o controle de alocação de espaço e sua própria formatação dos metadados, ou seja, o nome, o formato, o local onde estão armazenados, suas permissões de acesso, a data de criação, entre outros. Os que são mais usados atualmente são o EXT3, EXT4, NTFS, FAT, HFS, HPFS. Para fazer a distinção de qual significado estamos dando à expressão "sistema de arquivos"bastará analisar o contexto em que ela estará inserida.

Como estamos tratando de dados de computadores, não poderíamos deixar de falar dos componentes físicos que fazem a guarda destes. Neste capítulo, trataremos a respeito de componentes de armazenamento - os *hardwares* - conceitos gerais que envolvam sistemas de arquivos e uma visão superficial de como ele é organizado.

2.1 Mídias de Armazenamento

Os discos rígidos são de longe os componentes mais importante do computador quando tratamos de armazenamento de dados. Nesta sessão abordaremos a respeito de informações básicas sobre os discos e os métodos de acesso e escritas em blocos.

2.1.1 Discos Magnéticos

De acordo com Stallings [18] e Gutmann [7], um disco magnético é constituído de um prato de metal (ou de plástico) coberto com um material capaz de guardar estados magnéticos. Os dados são gravados e posteriormente lidos desse disco por meio de uma

bobina chamada cabeçote. Durante uma operação de leitura e escrita o cabeçote permanece estático enquanto o prato gira embaixo dele. A figura 2.1 mostra a parte interna de um disco.

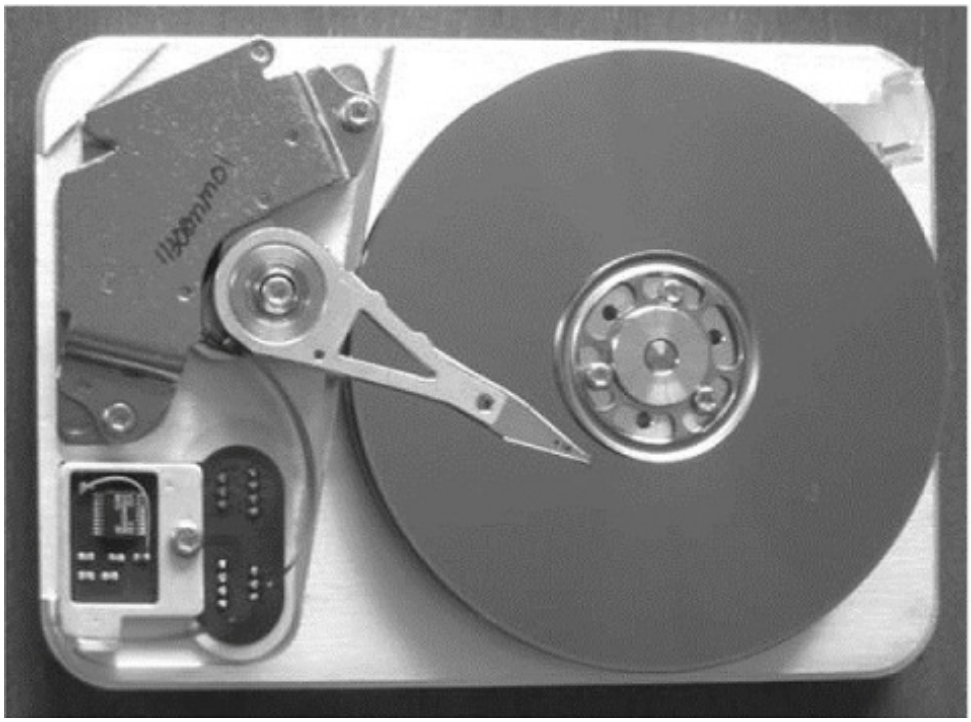

Figura 2.1: Parte interna de um disco ATA. Na direita estão os pratos e na esquerda o braço de leitura e escrita.Fonte [1]

O mecanismo de escrita é baseado no fato de que o fluxo de uma corrente elétrica que passa por uma bobina produz um campo magnético. Ao se enviar pulsos de correntes para o cabeçote, esses resultam em padrões magnéticos que são gravados na superfície embaixo dele; correntes positivas e negativas geram padrões magnéticos distintos. Já o mecanismo de leitura, se baseia no fato de que um campo magnético em movimento em relação a uma bobina, gera uma corrente elétrica nesta última. Logo, quando a superfície do disco se movimenta proximo ao cabeçote, gera uma corrente na bobina de mesma polaridade da usada na gravação, permitindo a recuperação dos dados.

Os dados são armazenados nos discos em blocos. Esses blocos são chamados de **setores**. Os setores concêntricos de um disco formam uma trilha. A cada trilha de um disco é associado um endereço iniciando da mais externa para o centro. Por exemplo, se temos 1000 trilhas em um disco e a mais externa seria a de número 0, a mais interna seria a 999. Devido a similaridade de todos os pratos e ao fato das trilhas em cada prato possuírem os mesmos respectivos números de endereço (prato 1 e prato 2, por exemplo, possuem ambos a trilha de numero 0, bem como a de número 999), existe um termo usado para se referir àquelas de mesmo número: **cilindro**. O cilindro 0 representa todas as trilhas de número 0 de cada prato. Conhecendo o cilindro onde uma informação está armazenada, bastaria mais dois dados para localizá-la no disco: o setor da trilha e a cabeça de leitura responsável pelo acesso (leitura ou escrita) aos dados. Em disco mais antigos o

método CHS (*Cylinder, Head, Sector*) era bastante usado no endereçamento de dados. Devido a limitações relacionadas as traduções de endereços na BIOS, ele foi praticamente abandonado. Os discos mais novos usam o método LBA (*Logical Block Address* - Endereçamento de Blocos Lógicos) que utilizam um único número (ao invés dos três anteriores) para referenciar os setores do disco [1].

Tanto entre os setores quanto entre uma trilha e outra existem espaços que além de diminuir erros devido a falta de alinhamento de cabeçotes ou da interferências de campos magnéticos servem para marcar o fim de um e o início de outro setor. Além desses erros, um setor pode se tornar defeituoso não garantindo a segurança dos dados neles armazenados e com isso não deve ser usado. Nos discos mais modernos a identificação desses setores é feita pelo próprio disco e os dados, neles contidos, são automaticamente remapeados para outras partes dos discos. Tanto os sistemas operacionais como também as controladora dos discos costumam guardar informações desses setores para evitar que dados sejam corrompidos.

2.1.2 Memórias Flash

De acordo com [3], as memórias flash são dispositivos eletrônicos não voláteis que podem ser apagados e reprogramados eletricamente. Existem dois tipos principais dessas memórias, conhecidas por *NOR* e *NAND*. Esses nomes estão relacionados com as características internas das células de memória flash individuais, as quais são semelhantes às das portas lógicas correspondentes.

As do tipo NOR chegaram ao mercado em 1988. Os chips possuem uma interface de endereços similar à da memória RAM e são muito usadas atualmente em BIOS de placas mãe e em *firmwares* de diversos dispositivos. O problema com as memórias do tipo NOR é o alto custo e, embora as leituras sejam rápidas, o tempo de gravação nas suas células é muito alto.

No caso das memórias do tipo NAND cada célula é composta por dois transístores, com uma fina camada de óxido de silício precisamente posicionada entre eles, que tem a capacidade de armazenar cargas negativas. Isso cria uma espécie de prisão para os elétrons e permite manter os dados por longos períodos de tempo, sem que seja necessário tanto manter a alimentação elétrica (como nas memórias SRAM), como dar cargas periódicas de energia (como na memória DRAM). Isso simplifica muito o design dos cartões, pendrives, SSD (*Solid State Drive* ou Unidade de Estado Sólido) e outros dispositivos, pois eles precisam incluir apenas os chips de memória Flash NAND, um chip controlador e as trilhas necessárias. Nada de baterias, circuitos de carga ou qualquer coisa do gênero.

Uma questão importante quando falamos de memórias flash é o fato de possuírem, até certo ponto, uma vida útil. Apesar de terem um número de leituras ilimitado, elas possuem um número de regravações pequeno, se comparados aos HDs.

Os chips baseados na tecnologia SLC (*Single-Level Cell*) suportam até 100.000 regravações. Os chips MLC(*Multi-Level Cell*), que são os mais usados, permitem apenas 10.000 processos de escrita. Esse número parece pouco mas quando pensamos em um SSD de 80 GB, teríamos de gravar 800 TB de dados para esgotá-lo. Se levarmos em conta que um usuário comum grava cerca de 40 GB por dia, ele levaria mais de 50 anos para esgotar a capacidade desse SSD. Teoricamente após esse prazo, não se poderia sobrescrever os dados contidos no SSD.

Os discos e as memórias flash representam os dispositivos principais de armazenamento. Pelas descrições percebemos que os dados nesses dispositivos são cargas elétricas ou campos magnéticos a nível físico. Dependendo de sua intensidade podemos interpretá-las como sendo 0's e 1's. Existem outros dispositivos que fazem uso dessa dualidade para a representação dos dados computacionais sem o uso da eletricidade, como os recentes estudos a respeito das propriedades químicas de materiais (*phase-change materials - PCM*) [8] feito por um grupo de pesquisa da universidade de Illinois (USA). Contudo, o importante para nosso estudo é sabermos que dados em computadores podem ser resumidos a dois *bits*: o 0 e o 1.

2.1.3 Abstrações

A abstração nos permite ignorar os detalhes internos de um grande sistema, e usá-lo como uma unidade compreensível. É por meio dela que dispositivos complexos são construídos. Aplicando a, torna-se possível estabelecer níveis de detalhamento diferentes conforme nossa necessidade. Em cada nível, o foco de operações é específico e muitas vezes não percebemos seus objetos finais. Mas quando olhamos o todo percebemos suas funcionalidades.

A interpretação dada aos *bits* na última sessão representa a abstração de nível mais baixo no sistema de arquivos. Ela é usada para se definir os *bytes*, como conjuntos de 0's e 1's totalizando 8 unidades de *bits*. Os *bytes* quando em conjunto, formam os setores (o padrão costuma ser 512 *bytes*). Esses por sua vez definem os blocos ou as unidades de dados no EXT3. Numerando os blocos é possível organizar todo o sistema de arquivos que veremos nas próximas seções.

Graças as abstrações, chegamos ao nível de um usuário clicar em um ícone, ou uma pequena imagem e ter a sua frente um documento pdf, ou um vídeo ou uma aplicação sem mesmo ter algum conhecimento do funcionamento de um disco. Não apenas em ciência da computação, mas de um modo geral, as abstrações são usadas para impulsionar o desenvolvimento de novas tecnologias bem como estimular a capacidade de criação dos seres humanos.

Além de ajudar na construção de diversos sistemas escondendo sua complexidade, as abstrações também são usadas nos sistemas de arquivos favorecendo sua performance e sua usabilidade. E é exatamente pelo seu uso que arquivos são excluídos sem necessariamente se tornarem perdidos ou inacessíveis. Peritos são capazes de recuperar arquivos muitas vezes excluídos a meses em computadores. Isso só é possível pois o que realmente é real são os campos magnético ou as cargas elétricas dos dispositivos físicos.

2.2 Volumes e Partições

A experiência mais comum relacionada a volumes de um sistema ocorre na instalação de um sistema operacional onde cria-se partições no disco rígido para posteriormente instalar o sistema. O processo de instalação guia os usuários através do processo de criação de partições primárias e lógicas, e no final o computador tem uma lista de "unidades"ou "volumes", para armazenar seus dados. Em ambientes de armazenamento de grande porte é comum também o uso de softwares de gerenciamento de volumes com o objetivo de ter vários discos de armazenamentos aparentando constituir um único grande disco.

Os volumes de um sistema estão baseados em duas ideias centrais. Uma é a de usar vários discos de armazenamento para criar um único volume e a outra é dividir os volumes em partições independentes.

Os termos "partição"e "volume"são freqüentemente usados juntos, mas é necessário fazer uma distinção entre eles. De acordo com Brian Carrier [1], o volume é uma coleção de setores endereçáveis de um sistema operacional que uma aplicação pode usar para armazenamento de dados. Os setores em um volume não precisam ser consecutivos nem pertencer a apenas um dispositivo de armazenamento físico, mas precisam dar a impressão de que eles o são. Um disco rígido é um exemplo de um volume que possuir setores consecutivos porém um volume pode também ser o resultado da união de discos. Já no caso das partições, elas fazem parte de um volume, podendo este possuir uma ou várias delas.

2.2.1 Partições

Uma partição é uma coleção de setores consecutivos de um volume. Por definição, uma partição também pode ser um volume, razão pela qual os termos são frequentemente confundidos. Caso seja necessário, vou me referir ao volume em que uma partição está localizada por volume da partição. As partições são utilizadas em vários cenários como:

- Alguns sistemas de arquivos têm um tamanho máximo que é menor do que os discos rígidos.
- Muitos *laptops* usam uma partição especial para armazenar o conteúdo da memória quando o sistema está ocioso.
- Sistemas UNIX usam partições diferentes para diferentes diretórios afim de minimizar erros por corrupção do sistema de arquivos.
- Sistemas baseados na plataforma IA32 que possuem múltiplos sistemas operacionais, como o Microsoft Windows e Linux, podem precisar de partições separadas para cada sistema operacional.

No sistema de partições é comum ter uma ou mais tabelas que os descrevem. Cada entrada da tabela representa uma partição. Nelas são armazenados os setores de início e fim da partição, e o seu tipo, como mostra a figura 2.2.

O objetivo da tabela de partição é o de organizar o *layout* de um volume. Portanto os únicos dados essenciais são o local de início e fim para cada partição. Na maioria dos casos, o primeiro e o último setor de uma partição não contém algo que os identifica como setores de fronteira. Por isso quando as estruturas dessa tabela estão corrompidas pode ser complicado realizar sua recuperação. Assim sistemas de grande porte, usam técnicas de montagem de volume para fazer vários discos ter a aparência de um. Além de aumentar a capacidade de armazenamento muitas vezes adiciona redundância aos dados, reduzindo o risco de falha de um disco.

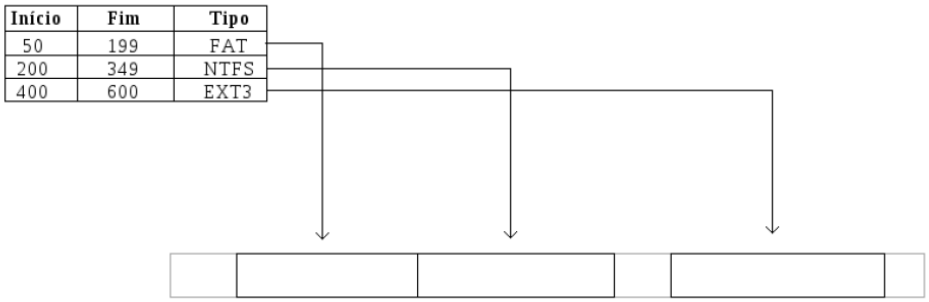

Figura 2.2: Uma tabela básica de partições mostrando o início, fim e tipo de cada partição. Adaptado de [1].

2.2.2 Endereçamento de Setores

O método mais comum para realizar o endereçamento de dados em um volume é o LBA. O endereço LBA é um número que inicia no 0, no primeiro setor do disco. Este endereço é o **endereço físico** de um setor.

Um volume é uma coleção de setores que necessita de endereços. Um **endereço lógico** de um volume é o endereço de um setor em relação ao início do seu volume. Logo, se considerarmos um volume como sendo um disco inteiro, o endereço físico será o mesmo que o endereço lógico. O início e o fim das partições de um volume são geralmente descritos usando endereços lógicos. Quando começamos a falar sobre o conteúdo de uma partição, existe uma outra camada de endereços lógicos. Estes endereços são em relação ao início da partição e não o início do volume ou do disco original. Se o setor não está alocado para uma partição, ele não vai ter um endereço lógico de volume, porém possuirá um endereço físico e um endereço de volume (que podem, ou não, ser o mesmo).

Assim o sistema localiza os setores utilizando esses endereços e enxergando o disco como um conjunto de setores contíguos.

A compreensão mais profunda de sistema de arquivos, principalmente com relação a estrutura de dados e programas relacionados a eles, tem ajudado muitos administradores de sistemas na condução de incidentes de segurança. Num mundo onde a informação ganha em importância, esse conhecimento aplicado a restauração de dados tem salvado tempo e dinheiro em várias empresas.

2.3 Arquivos

Os dados armazenados pelo sistema operacional não passam de 0's e 1's em nível físico. Afim de facilitar a vida dos usuários, o sistema operacional (SO) define uma unidade lógica de armazenamento denominada arquivos. De acordo com [19], podemos entender os arquivos usando dois pontos de vista. O primeiro é o do usuário e o segundo, o do programador. O usuário percebe a existência dos arquivos ao visualizar as pastas e ícones nas telas do computador. Já o segundo, o define como uma coleção de blocos contínuos de um disco, ou de um dispositivo de armazenamento do computador.

Normalmente, os dispositivos que guardam esses dados são não voláteis e com isso seu conteúdo persiste mesmo após quedas de energia ou reinicializações do sistema.

Existem diversos tipos de arquivos, da mesma maneira que existem diversos tipos de informações. Por exemplo, há os programas fontes, os objetos, os arquivos executáveis, os de textos, os de imagens, os de sons, até mesmo os diretórios , ou pasta, são arquivos (de um tipo especial que veremos mais adiante).

2.3.1 Atributos de Arquivos

Todo arquivo, quando criado, recebe um nome. Este é usado sempre que se deseje fazer referência a aquele. Os arquivos podem possuir diversos atributos, a depender do sistema operacional que se esteja utilizando. Normalmente os atributos mais comuns e que estão implementados nos sistemas atuais são os mostrados na tabela 2.1.

Tabela 2.1: Atributos de Arquivos

Atributo	**Descrição**
Nome	O nome de um arquivo costuma ser o único campo visível ao usuário.
Identificador	É um identificador único do arquivo. Costuma ser um número, o qual permite referenciar um arquivo em todo o sistema.
Tipo	É o tipo de arquivo. Por exemplo, se é um diretório, se é um arquivo de bloco, FIFO e assim por diante.
Localização	Pode ser um endereço para o local do arquivo no dispositivo ou um ponteiro para uma estrutura que guarda outras informações do arquivo.
Tamanho	O tamanho do arquivo (em *bytes*, ou blocos, ou *words*) e pode guardar o tamanho máximo permitido deste arquivo.
Controle de Acesso	Informações que determinam quem pode ter acesso ao arquivo no sentido de lê-lo, alterá-lo ou executá-lo.
Tempo, data, usuário	Guarda-se essas informações com relação a criação do arquivo, último acesso realizado e a última modificação efetuada. Esses dados muito usados quando se trata de segurança e monitoramento do sistema de arquivos.
flags	São *bits* de campos que controlam ou ativam propriedades específicas, como, por exemplo, a *flag* de arquivo oculto, que é usada para saber se o arquivo irá ou não aparecer em listagens de arquivos. Outro exemplo que podemos citar é a *flag* de arquivamento, usada para auxiliar os softwares de backup. Toda vez que um arquivo é alterado, essa *flag* é ativada e toda vez que o backup do arquivo é realizado, a *flag* é desativada.

As informações sobre todos os arquivos ficam armazenadas em uma estrutura de diretório e uma de metadados. Ambas são salvas em disco. Um diretório é uma lista de entradas as quais contêm, pelo menos, um nome e um endereço. Nos sistemas Unix ou *Unix-like*, por exemplo, usando esse endereço é possível localizar todos os atributos do

arquivo. Nos sistemas NTFS todas essas informações estão nas entradas da tabela mestra de arquivos (*MFT*).

Para definir um arquivo corretamente, é preciso considerar as operações que podem ser realizadas sobre eles. Os sistemas operacionais oferecem chamadas de sistema para criar, escrever, ler, reposicionar, apagar, truncar arquivos, dentre outras conforme mostra a tabela 2.2.

2.3.2 Estrutura de Arquivo

Arquivos podem ser estruturados de diversas maneiras. Três possibilidades muito usadas são representadas na figura 2.3.

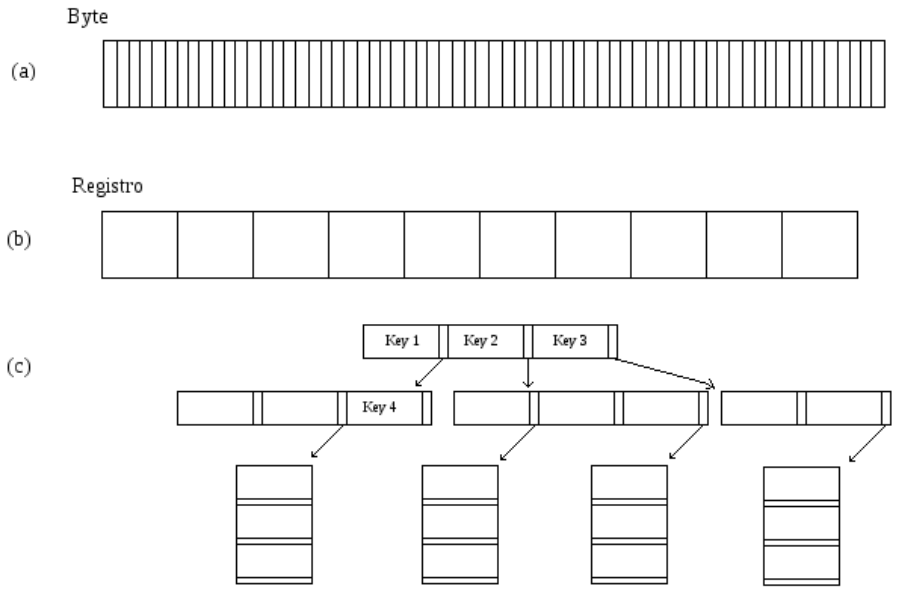

Figura 2.3: Representações de estruturas de um arquivo.

A primeira 2.3(a) representa os arquivos como uma sequência de bytes sem estrutura alguma. Dessa forma, qualquer significado dos dados são tratados por programas em nível de usuário, uma vez que o sistema operacional não se preocupa com o que há no arquivo. Isso da mais flexibilidade às aplicações e é a forma mais comum de se estruturar arquivos, principalmente nos sistemas usados por usuários comuns.

Na segunda representação 2.3(b) já se percebe alguma estrutura. O arquivo é visto como uma sequência de registros de tamanho fixo onde cada registro possui sua própria estrutura. Assim cada leitura do arquivo retorna um registro e cada operação de escrita anexa ou sobrescreve um registro.

A terceira 2.3(c) corresponde a uma árvore de registros. Cada um deles possui uma chave usada para facilitar buscas de registros dentro dos arquivo. Essa última forma de representação costuma ser usada em sistemas de computadores de grande porte voltados para o processamento de dados comerciais.

Tabela 2.2: Atributos de Arquivos

Operação	**Descrição**
Create	O arquivo é criado sem dados. O objetivo da chamada é para informar ao sistema operacional a respeito da existência do novo arquivo e que ele deve definir alguns dos atributos.
Delete	Quando o arquivo não é mais necessário, ele tem de ser excluído para liberar espaço em disco.
Open	Antes de usar um arquivo, um processo deve abri-lo. A finalidade dessa chamada é solicitar que o sistema operacional busque os atributos e a lista de endereços de dados do disco e carregue na memória principal para acesso rápido em chamadas posteriores.
Close	Quando terminam todas as operações que um processo deseja realizar sobre um arquivo e os seus atributos e os endereços de disco não são mais necessários o arquivo deve ser fechado para libertar algum espaço na tabela de descritores de arquivos e na memória. Muitos sistemas impõem um número máximo de arquivos abertos em processos. Além do mais um disco é escrito em blocos, e fechando o arquivo o sistema operacional é forçado a consolidar em disco as alterações feitas no arquivo.
Read	Os dados são lidos do arquivo. Normalmente, os *bytes* são lidos em sequências e existe um ponteiro que marca a posição de leitura corrente do arquivo. O processo que executa essa chamada deve especificar a quantidade de dados que devem ser lidos e também deve fornecer um *buffer* para guardá-los.
Write	Os dados são gravados no arquivo, mais uma vez, geralmente na posição corrente do ponteiro. Se essa posição é no fim do arquivo, o tamanho deste aumenta. Se a posição está no meio do arquivo, os dados existentes são substituídos pelos novos.
Append	Esta chamada é uma forma de gravação restrita. Só pode adicionar dados ao final do arquivo. Sistemas que fornecem um conjunto mínimo de chamadas de sistema geralmente não implementam o *append*, mas muitos oferecem várias maneiras de fazer a mesma coisa.
Seek	Para arquivos de acesso aleatório, é necessário um método para informar de onde os dados devem ser lidos. A chamada de sistema seek reposiciona o ponteiro do arquivo para um local específico do arquivo. Quando a chamada for concluída, os dados podem ser lidos ou escritos da posição escolhida.
Rename	Acontece freqüentemente um usuário precisar mudar o nome de um arquivo. Nem sempre essa chamada é necessária, porque os dados de um arquivo podem ser copiados para um novo com o novo nome e, o arquivo antigo apagado posteriormente. Porém muitos sistemas a utilizam.
Lock	Essa chamada bloqueia um arquivo ou parte dele impedindo o acesso simultâneo por diferentes processos. Para um sistema de reserva de passagens aéreas, por exemplo, bloquear o banco de dados ao fazer uma reserva evita a marcação de um mesmo assento por duas pessoas diferentes.
Set Attributes	Alguns dos atributos de arquivos podem ser alterados pelo usuário. Esta chamada de sistema torna isso possível. Alterações nas permissões de um arquivo são exemplos de uso desse tipo de operação. As alterações das flags dos arquivos também são exemplos nesta categoria.
Get Attributes	Processos muitas vezes precisam ler atributos de arquivo para realizar seu trabalho. Por exemplo, o programa make do UNIX é comumente usado para gerenciar projetos de desenvolvimento de software que possuem muitos arquivos de fonte. Quando ele é chamado, examina os tempos de modificação de todos os arquivos objeto fonte e faz o menor número de compilações necessárias. Assim, ele olha os atributos de cada arquivo.

2.3.3 Tipos de Arquivos

Existem diversos tipos de arquivos em um sistema. Podemos dividi-los em quatro grandes grupos. São eles os arquivos de usuários (os ditos de conteúdo), responsáveis pelo armazenamento de dados pertencentes aos usuários e os programas por ele usados; os arquivos do tipo diretório, que são arquivos que ajudam a criar as estruturas do sistema de arquivos; os arquivos especiais de caractere, relacionados com os dispositivos de entrada e saída; e por fim, os arquivos especiais de blocos usados para modelar os discos. Neste trabalho, nossa atenção estará voltada para os arquivos de usuário e os diretórios, tendo em vista que são os arquivos de maior importância quando estamos tratando de recuperação ou destruição de dados.

O conteúdo dos arquivos sugere que existam duas categorias, a saber: uma para os arquivos ASCII e outra para os binários. Estes últimos são caracterizados por possuírem uma estrutura interna com campos bem definidos de forma que os softwares que os utilizam possam reconhecê-los e utilizá-los em seus processos. Mesmo sendo constituídos por uma sequência de *bytes*, esses arquivos carregam consigo seções bem definidas. Eles só são executados pelos sistemas operacionais quando estão no formato correto. Se tentássemos imprimi-los provavelmente obteríamos algo ilegível.

Já os arquivos ASCII podem ser exibidos e impressos no formato em que se encontram, além de necessitarem apenas de um editor de texto comum para modificá-los.

2.3.4 Diretórios

Os sistemas de arquivos precisam de um mínimo de estrutura para funcionar corretamente. Para armazenar os dados necessários a esse funcionamento utiliza o disco. Parte dessas informações podem estar armazenadas em entradas de diretórios ou em estruturas de gerenciamento ou mesmo em estruturas de metadados. Os diretórios são uma das ferramentas usadas para dar base ao sistema de arquivos. Eles guardam listas de arquivos as quais permitem construir toda a árvore do sistema. Ele é constituído por entradas, uma para cada arquivo ou diretório, e cada uma delas possuem atributos e informações a respeito deles.

Se procurássemos nos dicionários de língua inglesa o significado da palavra diretório, compreenderíamos bem a sua função. De acordo com o Oxford [12], a palavra *directory* (diretório em inglês, ou se preferirmos uma tradução mais adequada: catálogo) é usada para se referir a um livro onde se encontra um lista de indivíduos, com nomes, endereços e telefones. Fazendo uma analogia aos diretórios dos sistemas de arquivos, esses indivíduos listados seriam as entradas (de cada diretório) e os dados dos indivíduos corresponderiam aos dados sobre os arquivos.

Existem sistemas operacionais que possuem estruturas específicas para armazenar as informações a respeito dos arquivos, como é o caso daqueles que utilizam as estruturas de *i-nodes*. Neles, as entradas de diretórios precisam conter apenas dois campos: um para guardar o endereço do *i-node* e outro para o nome do arquivo (conforme figura 2.4).

Quando é realizada uma requisição de abertura de um arquivo, o sistema operacional procura no diretório informado a entrada correspondente ao arquivo. Uma vez encontrada, extrai dela os atributos do arquivo ou informações que permitem localizar os blocos de conteúdo em disco. Nos sistemas que usam *i-nodes*, no lugar dos endereços dos blocos

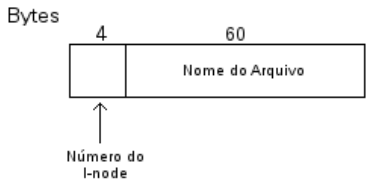

Figura 2.4: Exemplo de entrada de diretório do Unix.

de disco, haverá um ponteiro que leva a estrutura de *i-node* do arquivo, onde é possível conseguir os demais informações para carregar seu conteúdo.

Todo arquivo, para que possa ser manipulado, deve estar aberto. Para abri-lo, o usuário fornece o caminho para que o sistema operacional o localize. Primeiramente ele procura o diretório-raiz do sistema de arquivo. Este diretório pode estar em uma posição fixa no disco ou pode ser encontrado a partir de informações contidas nas estruturas do sistema. No UNIX clássico, por exemplo, o superbloco possui campos que guardam informações sobre a estrutura de dados do sistema de arquivos que precedem a área de dados. A partir do superbloco, encontramos os *i-nodes*. O primeiro *i-node* representa o diretório-raiz. Em sistemas operacionais que utilizam o sistema de arquivo do tipo NTFS, as informações do setor de inicialização permite localizar a MFT (Master File Table - Tabela de Arquivos Mestra). Essa tabela contém entradas para todos os arquivos/diretórios desse sistema e permite localizar qualquer outra parte do sistema de arquivos. Uma das 16 primeiras entradas dessa tabela é a "."(sem aspas) que contém um ponteiro para o diretório raiz.

Localizado o diretório-raiz, realiza-se uma pesquisa na árvore de diretórios até encontrar a entrada correspondente a informada pelo usuário. Ao encontrar a entrada de diretório que descreve o arquivo desejado, podemos encontrar diferentes informações a depender da maneira que os arquivos estarão organizados em disco pelo sistema operacional. Assim a entrada de diretório informará o endereço em disco que está o arquivo inteiro se for um sistema que usa a alocação contígua, ou o endereço do primeiro bloco se estiver usando listas encadeadas, ou ainda, o número do *i-node* ou da entrada em outra estrutura correspondente ao arquivo.

Concluímos portanto que é função dos diretórios proporcionar maior organização de dados para o usuário e mapear o nome ASCII de cada arquivo para as informações necessárias à localização deste no disco.

2.4 Layout de Sistemas de Arquivos

A maioria dos discos podem ser divididos em partições e cada uma delas podem conter sistemas de arquivos independentes. Qualquer partição pode guardar um ou mais sistemas operacionais.

De acordo com Andrew Tanenbaum [19], o setor "zero"de cada disco é conhecido como MBR ou *Master Boot Record* (Registro Mestre de Inicialização). Ele também possui outros nomes como *Initial Program Loader* (IPL) ou *Volume Boot Code* (VBC) ou ainda

masterboot. Usa-se esse setor para inicializar o computador. No final do MBR, existe uma tabela, onde é armazenada as informações de inicio e de fim de cada partição e qual delas é a ativa. Alguns sistemas operacionais não exigem que uma partição seja marcada como ativa oferecendo um menu ao usuário no qual escolha a que deseje inicializar.

Quando ligamos um computador, a BIOS lê e executa o código presente no MBR. A primeira ação desse programa será localizar a partição ativa desse disco. Posteriormente, lê e executar seu primeiro bloco (conhecido como *boot block* ou bloco de inicialização). Assim o bloco de inicialização carrega o sistema operacional da partição ativa e fazemos uso do computador.

Apesar das partições terem em seu *layout* o primeiro bloco destinado a carga do sistema operacional, as demais divisões do disco podem variar consideravelmente de um sistema para outro. Mesmo com as variações, é possível perceber que há a necessidade de existir uma região de controle do sistema como um todo, uma para o gerenciamento de espaços livres do disco, uma que organiza os arquivos desse sistema e uma para guardar o conteúdo dos arquivos. Um sistema de arquivo baseado no UNIX, por exemplo, contém os itens mostrados na figura 2.5.

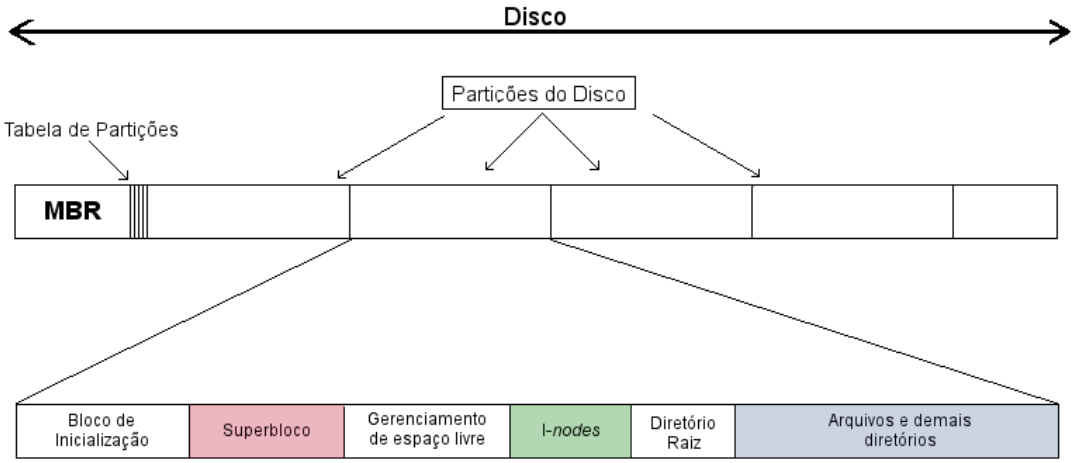

Figura 2.5: Layout de partição adaptado de [19].

O Superbloco é o local onde encontramos todos os parâmetros importantes sobre o sistema de arquivos. Ele é o bloco de controle e guarda as informações cruciais. É carregado em memória durante inicialização do sistema operacional.

Além do superbloco existe uma área responsável pelos blocos livres do disco. No caso dos sistemas UNIX também há uma região responsável por manter informações sobre os arquivos, inclusive onde seus blocos estão localizados no disco. Essa região do disco é composta por estruturas denominadas *i-nodes*. O diretório raiz marca o topo da árvore do sistema de arquivos e por fim o campo dos demais diretórios e arquivos do sistema.

Uma partição do tipo NTFS coloca todas essas regiões de controle em uma mesma tabela chamada tabela mestra de arquivos (MFT). De acordo com [11] essa partição possui uma estrutura semelhante a apresentada na figura 2.6.

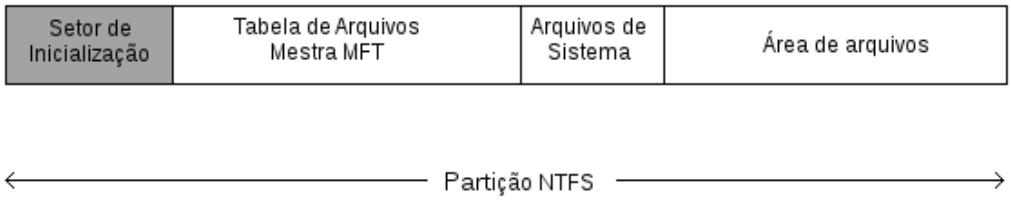

Figura 2.6: Estrutura de uma partição NTFS. Adaptado de [11]

2.4.1 Alocação Contígua

Certamente o ponto mais importante no armazenamento de arquivos é determinar quais blocos de disco corresponde a quais arquivos do sistema.

A alocação contígua é o sistema de alocação mais simples. Consiste em armazenar cada arquivo como uma sequência contígua de blocos do disco. É necessário saber apenas a posição do primeiro bloco e o tamanho do arquivo para carregá-lo. Logo, posicionando a cabeça de leitura no primeiro bloco, basta realizar operações de adição para encontrar os blocos restantes. O desempenho de leitura é excelente, já que é necessário apenas uma busca (*seek*) no disco. Por outro lado, com a sucessiva utilização do sistema, realizando exclusões de arquivos, passará a existir lacunas, gerando o que chamamos de fragmentação de disco.

Pode-se realizar, de tempos em tempos, a compactação do disco, porém essa é uma operação muito demorada. Outra opção para contornar esse problema da fragmentação seria manter uma lista dos espaços livres para tentar adequar o conteúdo dos novos arquivos àqueles. Contudo precisaria saber de antemão o tamanho final do arquivo.

Esse sistema foi usado durante muitos anos, mas devido aos problemas mencionados foi perdendo seu espaço. Com a criação das mídias óticas (CDs, DVDs) de gravação única os arquivos contíguos voltaram a ser utilizados.

2.4.2 Alocação Encadeada

Outro método de alocação e endereçamento de arquivos seria manter a sequência dos blocos como uma lista encadeada, onde a primeira palavra (*word*) de cada bloco seria um ponteiro para o próximo bloco.

Esse método resolve o problema das lacunas deixados no disco e não há necessidade de saber previamente o tamanho final do arquivo. Além disso, é necessário armazenar apenas a posição do primeiro bloco.

Por outro lado, o acesso aleatório em disco é extremamente lento. Para chegar ao bloco n seria necessário passar pelos $n-1$ blocos do arquivo, geralmente em posições não sequenciais. Em comparação com o método anterior haveria um custo maior de tempo para a leitura de um arquivo. Além disso, o volume de armazenamento de dados de cada bloco não seria uma potência de 2 devido ao ponteiro do início de cada bloco. Isso

não aparenta ser tão grave, contudo uma vez que programas de uma forma geral lêem e escrevem em blocos cujo tamanho é uma potência de 2, as leituras exigiriam obter e concatenar informações de pelo menos dois blocos, gerando sobrecarga.

2.4.3 Alocação Encadeada com Tabela em Memória

Para eliminar os dois problemas da alocação encadeada, criou-se uma tabela na memória principal que teria os ponteiros e endereço de blocos correspondentes a cada arquivo. Essa tabela recebeu o nome de FAT (File Alocation Table - Tabela de Alocação de Arquivos). A figura 2.7 exemplifica essa tabela.

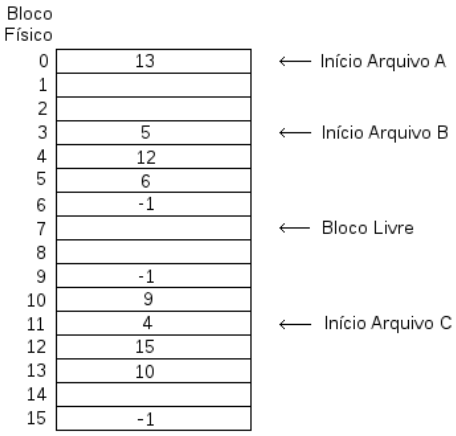

Figura 2.7: Alocação encadeada com tabela de alocação em memória principal.

Com essa alocação, não há a necessidade de reservar uma palavra de cada bloco para o ponteiro do bloco seguinte. Ainda existe a necessidade de percorrer o encadeamento para encontrar determinado deslocamento dentro do arquivo , mas é tudo feito em memória de forma que o custo para a operação é bastante reduzido.

Existe a necessidade de armazenar apenas o primeiro bloco do arquivo. Contudo esse esquema também guarda problemas. O principal deles é o tamanho da tabela FAT em memória. Além do fato dela ficar em memória o tempo inteiro, seu tamanho fica condicionado ao tamanho do disco já que cada uma de suas entradas está vinculada a cada *cluster* do disco. Considerando que cada entrada dessa tabela precise de 3kb a 4kb de espaço no minimo, um disco de $20GB$ com tamanho de bloco de $1k$ precisaria de $60MB$ a $80MB$ de memória principal. De acordo com [15], considerando ainda que esta tabela é vital para o funcionamento do sistema inteiro duas cópias dela são mantidas em disco.

2.4.4 I-node

Conforme [13], *index node* ou *i-node* é uma estrutura de dados que armazena informações sobre um arquivo. Cada arquivo fica associado a uma dessas estruturas, a qual é identificada por um número inteiro (*i-node number* ou *i-number*). Um *i-number* se

comporta como índice em uma tabela de *i-nodes* no sistema operacional. Ele fica armazenado na entrada de diretório relacionada ao arquivo. Assim, quando há requisição de abertura de um arquivo uma estrutura semelhante a mostrada na figura 2.8 é carregada em memória.

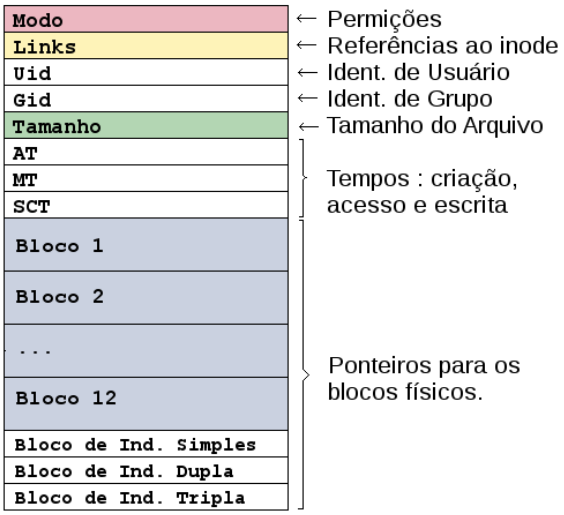

Figura 2.8: Estrutura do I-node.

Com esse método, não é necessário manter uma tabela (como a FAT) com as informações de todos os arquivos do sistema. Basta carregar o *i-node* do arquivo que será aberto. A ocupação em memória nesse caso fica ligada a quantidade de arquivos abertos e não mais ao tamanho do disco.

No início existia um problema relacionado a quantidade de endereços de blocos de disco que um *i-node* conseguiria guardar, mas isso foi resolvido ao reservar os três últimos campos do *i-node* para apontar para outros conjuntos de endereços. Esses são os chamados ponteiros de indireção simples, dupla ou tripla. Os demais campos dos *i-nodes* são mostrados na figura 2.8.

Os campos de principal interesse neste trabalho são os de endereço de conteúdo dos arquivos. No desenvolvimento de sistemas de arquivos, um dos princípios usados é o da localidade. Quanto mais próximo os dados estiverem, menos tempo o disco gasta para disponibilizá-los em memória. Quando o sistema operacional precisa de espaço em disco para um arquivo, ao invés do sistema alocar cada bloco de disco, ele aloca um conjunto de blocos com o intuito de manter os dados desse arquivos o mais próximo possível. Nos *i-nodes*, conforme mostra a figura 2.8, os campos que armazenam dados dos endereços dos blocos de conteúdo são os denominados Bloco 1, Bloco 2 ....

2.4.5 Superblocos

O Superbloco é uma estrutura que armazena informações as quais descrevem e mapeiam o *layout* do sistema de arquivos. Ele contém dados básicos, tais como o tamanho

e o número total do blocos, o número de blocos por grupo e quais são os reservados - antes do primeiro grupo de blocos-, número total de *i-nodes* e quantos são por grupos. Contém alguns dados necessários à administração do sistema como o número total de *i-nodes* e blocos livres, local de montagem do sistema de arquivo, valores que identificam se o sistema de arquivos está consistente entre outros dados. Ele também guarda dados não essenciais como o nome do volume, o último tempo de escrita em disco, a última vez que foi montado, etc.

Dessa forma, os superblocos desempenham um papel fundamental na organização dos arquivos e na estruturação dos sistemas que os utilizam (os baseados no UNIX). Devido a sua importância, esses sistemas costumam guardar cópias dessa estrutura em outras partes do disco podendo recuperá-la em caso de erros nnos seus dados.

2.4.6 Monitorando Blocos Livres

Considerando que todo sistema possui seu disco dividido em blocos é necessário manter quais desses blocos estão livres e quais podem ou não receber dados.

Na fabricação de dispositivos de armazenamento é praticamente inviável construir discos 100% perfeitos. Para não precisar destruir e refazer os discos, geralmente as controladoras mantêm uma lista de todos os blocos defeituosos e impedem que esses setores sejam usados para arquivamento. Existem muitos software que tentam evitar o uso de setores problemáticos mapeando-os e marcando-os como regiões ocupadas.

Uma das maneiras de se monitorar o uso de blocos é por meio de uma lista encadeada. Ela guardaria o endereço dos blocos livres do disco. Logo, quanto mais cheio o disco estiver, menor a lista será. Se cada bloco de disco possui 1KB de espaço, poderemos guardar 255 números de blocos de 32 bits por bloco. O que significa que seria necessário no mínimo pouco mais de 1GB para armazenar 256GB de espaço livre em disco. Essa lista encadeada costuma ser armazenada nos blocos livres do disco, não impactando tanto o uso de memória RAM. Os sistemas operacionais costumam manter em memória apenas a posição do próximo bloco livre agilizando o processo de alocação.

Outra maneira de fazer mapeamento de blocos livres é utilizando bits 0's e 1's para simbolizar blocos ocupados e vazios (ou vice-versa). Essa técnica é conhecida como **Mapa de bits**.

2.4.7 Criação de Arquivos

Dois passos são necessários para a criação de um arquivo. O primeiro é encontrar espaço suficiente no sistema para o arquivo para o seu conteúdo. O segundo trata-se da criação de uma nova entrada no diretório onde o nome do arquivo e o ponteiro para o seu conteúdo irá residir.

2.4.8 Manutenção de Arquivos

Para realizar a leitura de um arquivo, usamos uma chamada de sistema que especifica o nome do arquivo e o lugar da memória onde o próximo bloco de arquivo deve ser inserido. Assim, realiza-se uma pesquisa no diretório pela entrada associada ao arquivo. O sistema mantém um ponteiro de leitura para o local onde será realizada a próxima leitura.

Para escrever num arquivo, realiza-se outra chamada de sistema passando como parâmetro o nome do arquivo juntamente com os dados a serem escritos neste. O sistema por meio do diretório localiza o arquivo e usando o ponteiro de escrita, realiza a atualizando do arquivo.

2.4.9 Exclusão de Arquivos

A exclusão de um arquivo possui efeitos diretamente visíveis aos usuários. No momento seguinte a essa ação, o arquivo já não aparece nas listagens de diretórios. A nível de sistema de arquivos as informações não desaparecem. A depender de cada sistema de arquivos, a remoção terá tratamentos diferentes. Alguns apenas marcam os arquivos como prontos para serem removidos ocultando seu nome outros sobrescrevem algumas estruturas para registrar o estado de remoção. Mas na maioria das vezes o conteúdo mantido no sistema ainda permite sua recuperação.

De acordo com [20], quando um arquivo é removido em um sistema UNIX, por questões de performance, procura-se realizar a menor quantidade de alterações no sistema de arquivos.

1. A entrada de diretório com o nome do arquivo é marcada como não utilizada, de modo que o nome do arquivo se desvincula de todas as informações do arquivo (*inodes*). Os nomes de arquivos removidos continuam nas entradas de diretórios até serem substituídos por outros.

2. O bloco do *inode* é marcado como não utilizado em seu mapa de bits de alocação de blocos. Algumas informações de atributo de arquivo são sobrescritas. Sistemas mais antigos podem manter até mesmo os blocos de endereço de conteúdo intactos. Isso mantém as conexões do *inode* com o conteudo do arquivo vivas, o que facilita a sua recuperação. Em particular, no caso da distro Opensuse 12.3, os endereços guardados no *i-node* do arquivo excluído é sobrescrito com zeros.

3. Blocos de conteúdo de arquivos são marcados como livres no mapa de *bits*, no entanto os seus conteúdos permanecem no disco. Nos sistemas de arquivos mais recentes os blocos de conteúdo perdem a conexão com o *i-node* mas suas informações ainda são encontradas em arquivos journals, como veremos no próximo capítulo.

A figura 2.9 resume as informações que normalmente são preservada e destruídas quando um arquivo é apagado.

Informações do Arquivo	Local de Armazenamento	Efeito da Exclusão
Nome	Diretório	Presevado, disconectado do arquivo
Atributos	I-node	
Proprietário		Prevervado
Grupo		Prevervado
Tempos de Leitura/Escrita		Prevervado
Tempo de última alteração de atributos		Tempo da Exclusão
Tempo de Exclusão		Tempo da Exclusão
Contador de referências de diretórios		Zero
Tipo de Arquivo		Linux: preserva; Outros: destroem
Permissões de acesso		Linux: preserva; Outros: destroem
Tamanho do arquivo		Linux: preserva; Outros: destroem
Endereços de blocos de disco		Linux: preserva; Outros: destroem
Conteúdo	Blocos de Disco	Linux: preserva; Outros: desvinculam

Figura 2.9: Resumo das informações do *i-node* após exclusão de arquivos. Com adaptações de [5].

Capítulo 3

Por dentro dos Sistemas de Arquivos

Neste capítulo aprofundaremos os conhecimentos a respeito de um dos principais sistemas de arquivos usados atualmente: o EXT3.

Nele detalharemos suas estruturas baseando-se principalmente no livro escrito por Brian Carrier [1] mostrando o seu funcionamento na alocação e na exclusão de um arquivo.

3.1 Ext2/3

Os sistemas de arquivos Ext2 e Ext3, os quais chamaremos como ExtX, juntamente com o novo Ext4 são os sistemas de arquivos padrão na maioria dos sistemas operacionais Linux. O Ext4 é a mais nova versão dos ExtX e propõe algumas alterações de *layout*, expansão das capacidades relacionadas aos arquivos (como tamanho de um único arquivo e espaço total de um sistema) mas por ser uma evolução dos anteriores, guarda consigo muitas marcas dos sistemas ExtX.

Os dados que descrevem o *layout* de um sistema de arquivo ExtX são armazenados em uma estrutura chamada **superbloco**, localizada no início de cada grupo de blocos do sistema de arquivos. O conteúdo de cada arquivo é salvo em blocos. Os blocos podem ser descritos como um conjunto de setores consecutivos.

Os atributos de cada arquivo são armazenados nos *i-nodes*, os quais são estruturas que possuem um tamanho fixo e são organizados em tabelas. Existe uma tabela de *i-node* em cada grupo de blocos. O nome dos arquivos são armazenados nas entradas de diretórios. Estas estruturas são organizadas de maneira simples que guardam além do nome, um ponteiro para seu *i-node* correspondente.

Nas próximas subseções detalharemos as partes do sistema de arquivos, dividindo-as de acordo com suas características mais marcantes.

3.1.1 Dados Administrativos

No ExtX, existem duas estruturas que armazenam dados sobre o sistema de arquivos: o superbloco e o descritor de grupo. O superbloco possue informações a respeito dos tamanhos do arquivo e as configurações do sistema. Ele é organizado em grupos de blocos e cada grupo possui uma estrutura que o descreve. Os descritores de grupos são armazenados em uma tabela localizada imediatamente após o superbloco. Devido a

importância desses dados para o sistema, cópias de segurança são mantidas em partes diferentes do sistema.

Superbloco

Está localizado a 1024 *bytes* do início do sistema de arquivos e costuma ocupar 1024 *bytes* de espaço em disco.

Ele guarda o tamanho dos blocos, o número total de blocos, o número de blocos por grupo, número de blocos reservados antes do primeiro grupo de blocos, o número total de *i-nodes* e a quantidade de *i-nodes* por grupo de blocos. Algumas informações não essenciais podem também ser guardadas neles como: nome do volume, último tempo de escrita, última vez que foi montado, dados de checagem de consistência dos dados, números de *i-nodes* e blocos livres para alocação imediata.

Assim o superbloco define as características do sistema de arquivos. Devido a sua importância, cópias de segurança do superbloco pode aparecer no inicio de qualquer grupo de blocos.

Tabelas de Descrição de Grupos

Depois do superbloco, os sistemas ExtX armazenam a tabela de descritores de grupos. Além do conteúdo de cada arquivo, os grupos de blocos contém dados administrativos (superbloco e tabela de descritor de grupo), tabela de *i-nodes*, os mapas de bits dos i-nodes e os mapas de bits dos blocos de conteúdo conforme imagem 3.1. Os descritores de grupos são responsáveis por mapear onde cada um desses componentes estão localizados.

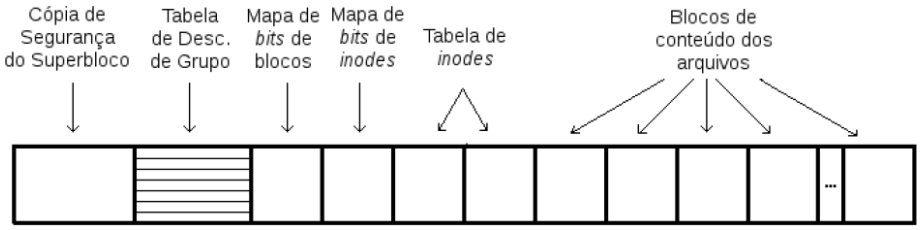

Figura 3.1: *Layout* de um grupo de blocos de um sistema ExtX. Adaptado de [1].

Os mapas de *bits* são usados para gerenciar o estado de alocação dos componentes do sistema de arquivos. Usando 0's e 1's é possível dizer se um *i-node* ou um bloco de dados está reservado ou não. Os descritores de grupos mantém informações qua agilizam a tomada de decisão do sistema, com isso mantém atualizado quantos *i-nodes* e quanto blocos do grupo estão livres. A sua estrutura é dada na tabela 3.1.

Código de Inicialização

O código de inicialização pode ou não existir em um sistema de arquivos ExtX. Caso o *kernel* do sistema operacional esteje gravado na partição ExtX, temos duas possibilidades de localização do código de inicialização. Ou nos dois primeiros blocos do sistema

Bytes	Descrição
0 - 3	Endereço inicial do bloco de mapas de bits dos blocos de conteúdo.
4 - 7	Endereço inicial do bloco de mapas de bits dos *i-nodes*.
8 - 11	Endereço inicial da tabela de *i-nodes*.
12 - 13	Número de blocos não alocados no grupo.
14 - 15	Número de *i-nodes* livres no grupo.
16 - 17	Número de diretórios no grupo.
18 - 31	Não utilizados.

Tabela 3.1: Campos de um descritor de grupo de um sistema de arquivo ExtX. Adaptado de [1]

(imediatamente antes do superbloco) ou na MRB (*Master Record Boot*) usando um *boot loader*.

O código de inicialização é usado para para carregar o *kernel*. Logo, se ele não existe, o espaço de 1024 *bytes* antes do superbloco fica preenchido com zeros.

3.1.2 Conteúdo dos Arquivos

Os sistemas de arquivos ExtX dividem o disco em blocos (ou se preferirem, um conjunto de setores consecutivos) para o armazenamento de dados. A ideia é similar ao conceito de *clusters* do FAT e do NTFS.

Blocos

Os blocos costumam ter 1KB, 2KB ou 4 KB de tamanho. Todo bloco possui um endereço. Ele inicia do 0, no primeiro bloco do sistema de arquivos, e incrementa até encontrar o último. Os blocos são agrupados de modo a termos uma zona (grupo de blocos), com excessão dos blocos pertencentes a área reservada do superbloco (neste caso, o primeiro grupo de blocos vem após essa área). É no superbloco que está definido a quantidade de blocos por grupo e o tamanho em *bytes* por bloco.

Estado de Alocação dos Blocos

Como vimos na seção de descritores de grupos 3.1.1, o estado de alocação de cada bloco é definido pelo mapa de *bits* do grupo de blocos ao qual ele pertence. Cada *bit* do mapa de blocos corresponde ao estado de alocação de um bloco dentro do grupo. Para determinar qual *bit* pertence ao bloco, é necessário saber o endereço do primeiro bloco, o endereço do bloco que se deseja obter o estado de alocação e calcular em qual *byte* esse *bit* correspondente está no mapa de *bits*. Depois basta usar a operação *AND* com uma mascara de 8 *bits* para saber seu estado de alocação.

No Linux, a alocação de blocos é feita baseada no grupo de blocos, mas pode acontecer de haver outras implementações a depender dos sistemas operacionais. Quando um bloco é alocado para um *arquivo*, o *linux* procura o primeiro bloco disponível no mesmo grupo do *i-node* do arquivo. Dessa forma ele tenta reduzir a quantidade de movimento da cabeça de leitura do disco ao ler um arquivo. Caso seja necessário expandir o tamanho de um

arquivo, ele irá buscar, iniciando pelo final deste arquivo, blocos usando a estratégia do primeiro disponível. Está implementado também um recurso do sistema de arquivos que faz uma pré-alocação de blocos no caso de diretórios para que os arquivos evitem de se tornar fragmentados, quando necessitem de se expandir. Caso não exista mais espaço em um grupo de blocos para a expansão de um arquivo, o Linux usa um bloco de um outro grupo.

3.1.3 Metadados

Os metadados dos arquivos em um sistema de arquivos ExtX estão guardados nos *i-nodes*.

I-nodes

Todos os *i-nodes* do sistema possuem um tamanho definido no superbloco. Há um *i-node* para cada arquivo do sistema sendo que cada um possui um endereço único, começando do número 1. Um conjunto de *i-nodes* é reservado para cada grupo de blocos (zonas) e são armazenados em uma tabela (tabela de *i-nodes*) cuja localização é dada pelo descritor de grupos.

Normalmente os sistemas de arquivo ExtX reservam os 10 primeiros *i-nodes*. Costuma-se usar o de número 1 para guardar os setores de disco que estão danificados. Mas apenas o de número 2 tem função específica: armazenar dados do diretório raiz do sistema.

Os *i-nodes* possuem campos fixos podendo usar o campo de extensão de atributos ou os ponteiros para blocos indiretos para guardar informações adicionais. O estado de alocação de cada *i-node* é determinado pelo mapa de *bits* dos *i-nodes*.

Um *i-node* é colocado em estado de não alocado, quando não há mais nenhum processo com o arquivo aberto e o número de referências (*links*) para o arquivo que ele representa chega a 0.

Tudo no sistema *Unix* é um arquivo. A definição do tipo a que um arquivo pertence é obtida pelo atributo *mode*. Com isso o Unix possui os arquivos que mapeiam os dispositivos de blocos (como os discos, por exemplo), os arquivos que representam os dispositivos de caracteres (como os teclados), também chamados de *raw device*, os usados para troca de dados entre processos (FIFO ou também conhecidos como *pipes* e os *sockets*), os de referência a outros arquivos como os *links* e por fim os arquivos e diretórios de usuários os quais poderíamos chamar de arquivos regulares.

Ponteiros para os Blocos

Tanto os sistemas ExtX como o UFS foram projetados para serem eficientes com arquivos pequenos. Portanto, cada *i-node* armazena 12 ponteiros de endereçamento direto de conteúdos do arquivo em sua estrutura, o que proporciona um acesso rápido a arquivos pequenos. Caso o arquivo tenha tamanho maior que 12 blocos, um novo campo do *i-node* é reservado para armazenar um ponteiro para os endereços restantes dos blocos de conteúdo do arquivo. Os ponteiros para os blocos que irão guardar os endereços de acesso ao conteúdo do arquivo são chamados de ponteiros de endereçamento indireto. Para conseguir lidar com arquivos maiores ainda, existem também os blocos de endereçamento duplo e triplo. O primeiro é um ponteiro para um bloco que guardam endereços de blocos

de endereçamento indireto. E o segundo, por sua vez, é um ponteiro para um bloco que possui endereços que levam a blocos de endereçamento duplo, como mostra a figura 3.2. Isso proporciona um tamanho máximo de um arquivo entre 16GB e 2TB.

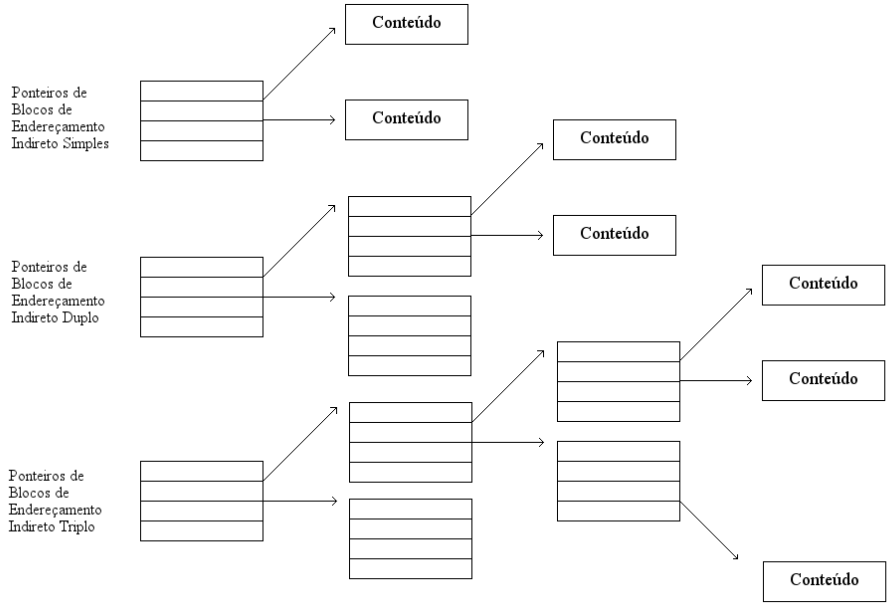

Figura 3.2: Ilustração de ponteiros de endereçamento indireto simples, dublo e triplo um sistema ExtX. Adaptado de [1].

Alocação de *I-nodes*

Quando um *i-node* é alocado, seu conteúdo é apagado, e os tempos do arquivo são atualizados para a hora atual. A quantidade de *links* para o arquivo, ou seja, aquilo que identifica quantos nomes de arquivos estão apontando para o *i-node*, é definido como 1. Diretórios terão o valor 2 atribuído a esse campo por causa da entrada '.' dentro dele.

Quando um arquivo é excluído no Linux, o valor armazenado no campo *link* no *i-node* é decrementado por um. Se este valor chega a zero, então o *i-node* é desalocado. Se um processo mantém o arquivo aberto, então este se torna um arquivo órfão e é adicionado em uma lista no superbloco para futuro tratamento. Quando o processo fecha o arquivo ou quando o sistema for reinicializado, o *i-node* será liberado para nova alocação.

De acordo com [1], o Ext2 e Ext3 lidam de forma diferente com relação a o que ocorre no *i-node* na exclusão de um arquivos. Em suas pesquisas no Fedora Core 2, o sistema de arquivo Ext3 atribui 0 ao tamanho do arquivo e limpa os ponteiros de blocos diretos e indiretos do *i-node*. O Ext2, por outro lado, não limpa esses valores, o que torna a recuperação de arquivos mais fácil. O M-tempo (tempo de modificação), C-tempo(tempo de modificação de metadados), e D-tempo(tempo de exclusão do arquivo) também são atualizados para refletir o tempo da exclusão. Se o arquivo foi excluído porque ele foi transferido para um volume diferente, o seu A-tempo (tempo de acesso ao conteúdo do arquivo) também será modificado para quando o conteúdo foi acessado. Isto poderia

ajudar a distinguir entre uma exclusão e uma mudança de diretório do arquivo (comando `mv`).

Entradas de Diretórios

Como vimos anteriormente, os diretórios contém uma lista de entradas que possuem o nome de um arquivo e o endereço de *i-node* correspondente a ele. Todo diretório inicia com duas entradas ('.' e '..') que fazem referência, respectivamente, ao diretório corrente e ao pai. O seu tamanho corresponde a quantidade de blocos alocados para suas entradas.

Cada entrada possui um tamanho dinâmico, a depender do comprimento do nome de cada arquivo (1 a 255 *bytes*). Existe um campo em cada entrada que identifica esse comprimento (sempre múltiplo de 4). Há também um campo para identificar o tipo de arquivo.

A estrutura das entradas de diretório pode variar de acordo com o sistema operacional. No geral possui os campos mostrados na tabela 3.2.

Bytes	Descrição
0 - 3	Referência ao *I-node*.
4 - 5	Comprimento da entrada.
6 - 6	Comprimento do nome do arquivo.
7 - 7	Tipo de arquivo (arquivo regular, de bloco, FIFO, diretório...).
8 +	Nome do arquivo em ASCII.

Tabela 3.2: Estrutura de dados de uma entrada de diretórios. Adaptado de [1]

Árvores de *Hash*

Nos sistemas ExtX as entradas de diretório podem está organizadas na forma de uma árvore de *hash* ou simplesmente adicionadas (sem nenhuma organização prévia) no momento que são geradas. Existe uma *flag* no superbloco que evidencia a forma de organização escolhida na instalação do sistema operacional.

Essa árvore de *hash* é semelhante a árvore B usada pelo NTFS. A grande diferença está no fato da primeira ordenar seu conteúdo baseado no *hash* do nome do arquivo e a segunda é baseada no nome dele.

Caso um diretório utilize as árvores de *hash*, ele será constituído de vários blocos onde cada um deles representa um nó da árvore. O primeiro bloco será composto das duas entradas principais ('.' e '..') e no espaço restante conterá os descritores dos nós da árvore os quais são usados durante a busca por arquivos.

3.1.4 Monitorando Modificações no Sistema

O Ext3 inclui um sistema de *log* de arquivos. Ele se assemelha a um diário do que acontece no sistema de arquivos. É responsável por registrar as atualizações feitas no sistema. Isso permite que ele se recupere mais rapidamente caso acontece algum acidente com relação aos dados. Esse sistema é chamado de *journal*.

O arquivo *journal* do Ext3 costuma utilizar o *i-node* de número 8. Ele é considerado um recurso do sistema de arquivo podendo o seu uso ser habilitado, ou não, de acordo com o valor armazenado nos *bytes* 208 a 231 do superbloco.

Carrier [1] nos mostra que o *journal* guarda os blocos sofreram atualização em algum processo, e depois que ela sensibiliza o disco, ele identifica sua conclusão. Existem dois modos pelos quais o *journal* pode operar. No primeiro, são registrados apenas os blocos de metadados que serão alterados. No outro, acrescenta-se também os blocos de conteúdo nos seus registros de transações. A primeira versão do *journal* usou essa última abordagem. Mas por questões de crescimento demasiado do arquivo a versão atual registra apenas as alterações de metadados.

O *journal* no Ext3 é feito a nível de bloco, o que significa dizer que se um *bit* de um *i-node* for alterado, o bloco inteiro em que o *i-node* está localizado será salvo no *journal*. Seu primeiro bloco é reservado para manter informações estruturais e de gerência desse sistema. Os outros blocos são usados para registrar as transações que ocorrem no sistema. Elas aparecem na forma de as entradas do *journal*.

Cada transação realizada sobre arquivos recebe um número sequencial. As transações possuem um descritor de bloco composto do número sequencial e os blocos do sistema que serão atualizados. A frente do bloco descritor, ficam os blocos de metadados que sofreram a atualização. Quando a atualização atinge o nível físico (são efetivamente escritas em disco) passa a existir um bloco de confirmação com o mesmo número sequencial da transação. Depois do bloco de confirmação inicia uma nova transação.

Pelo fato de armazenarem informações dos *i-nodes* antes de sua destruição, o *journal* pode ser utilizado na recuperação de arquivos pois os endereços de conteúdos destes podem ser encontrados nas transações registradas por aquele.

Funcionamento do Sistema de Arquivos

Depois de visualizar os componentes do sistema de arquivos ExtX, veremos a seguir como todas essas estruturas e dados se juntam na prática. Primeiramente, iremos alocar um arquivo de 6000 *bytes* em um sistema ExtX com tamanho de bloco de 1024 *bytes*.

Alocação de Arquivos

O processo de criação do arquivo `/dir1/file1.321` inicia na localização do diretório `dir1`. Depois cria-se uma entrada de diretório nele, aloca um *i-node* para o arquivo e aloca-se os blocos para o seu conteúdo. Logo temos os seguintes passos:

1. O sistema lê a estrutura de dados superbloco, que está localizada no *bytes* 1024 do sistema de arquivos e o processa afim de conseguir o tamanho de cada bloco (1024 *bytes*). Cada grupo de blocos tem um total de 8192 blocos e 2016 *i-nodes*. Não há blocos reservados antes do início do primeiro grupo de blocos.

2. Em seguida o sistema lê a tabela de descritores de grupo afim de definir o *layout* de cada grupo de blocos.

3. Como precisamos localizar o diretório "`dir1`"no diretório raiz, então o sistema processa o *i-node* 2 (*i-node* do diretório raiz). Usando o número de *i-nodes* por grupo , determinamos que *i-node* 2 está no grupo 0. Então o sistema usa a entrada da

tabela de descritor de grupo para achar o bloco correspondente à tabela de *i-nodes* do grupo 0. Assim ele descobre que ela está no bloco 6.

4. Ele faz a leitura da tabela de *i-node* que está no bloco 6 e processa a segunda entrada. O *i-node* do diretório raiz mostra que o seu conteúdo, suas entradas de diretórios, estão localizadas no bloco 258 do disco.

5. Realiza-se então a leitura do conteúdo do diretório raiz, que está no bloco 258, e processa o conteúdo como uma lista de entradas de diretórios. As duas primeiras entradas são para '.' e '..'. O sistema operacional (SO) avança para a nova entrada afim de encontrar a de nome `"dir1"`. Ao encontrá-la descobre que o *i-node* correspondente a ela é o de número 5033. O tempo de acesso do diretório raiz é atualizado.

6. É necessário agora encontrar onde o *i-node* 5033 está localizado. Ao se dividir esse número pelo número de *i-nodes* por grupo percebemos que ele se encontra no grupo de bloco de número 2. Usando a entrada do descritor de grupo do grupo 2, determinamos que sua tabela de *i-node* começa no bloco 16390.

7. Realiza-se a leitura da tabela de *i-nodes* do bloco 16390 e processa a entrada 1001, que é o localização relativa do *i-node* 5033. O conteúdo do *i-node* revela que o conteúdo de `"dir1"`está localizado no bloco 18431.

8. O SO faz a leitura do conteúdo do `"dir1"`no bloco 18431 e processa a lista de entrada de diretório para localizar algum espaço não utilizado no diretório. O nome é adicionado entre dois nomes alocados ou no final dessa lista, e os tempos de modificação de metadados e modificação do conteúdo do diretório são atualizados. As alterações de conteúdo de diretório são registradas no arquivo *journal*.

9. Após alocar a entrada de diretório, é necessário alocar um *i-node* para o arquivo, o qual será alocado no mesmo grupo de blocos do diretório pai desse arquivo, ou seja, grupo 2. O SO procura e encontra o mapa de *bits* dos *i-nodes* para esse grupo no bloco 16386 utilizando o descritor de grupos. Busca o primeiro *i-node* livre para fazer a alocação e encontra o de número 5110 vazio. Seu *bit* correspondente no mapa de *bits* é definido como 1, decrementa o número de *i-nodes* livres na tabela de descritor de grupo. Adiciona o endereço de *i-node* na entrada de diretório recém criada. E os blocos que sofreram mudanças durante o processo (mapa de *bits*, descritor de grupo e superbloco) são gravados no *journal*.

10. O conteúdo do *i-node* 5110 é preenchido com os dados do novo arquivo. Assim valores de tempo são registrados e é atribuído o valor 1 para o número de *links* do arquivo. As novas mudanças também são registradas no arquivo *journal*.

11. Chegando ao final do processo, verifica-se a necessidade de se alocar seis blocos para armazenar o conteúdo do arquivo. O SO processa o mapa de *bits* e localiza os blocos 20002, 20003 e de 20114 a 20.117 . Os *bits* referente a cada um destes blocos é definido como 1 e os endereços são usados para preencher os campos de ponteiros de conteúdo de acesso direto do *i-node* 5110. Tanto o contador de blocos livres do descritor de grupo quanto os do superbloco são atualizados. Os tempos modificação do *i-node* e de mudança de conteúdo são atualizados. As alterações feitas nos blocos

de *i-nodes*, do descritor de grupo, do superbloco e do mapa de *bits* são registradas no *journal*.

12. Finalmente, o conteúdo do arquivo file1.321 arquivo é escrito nos blocos alocados.

Exclusão de Arquivos

Aproveitando o exemplo anterior, vamos agora excluir o arquivo que acabamos de alocar. A ordem de desalocação das estruturas do sistema de arquivo pode variar de acordo com a implementação do sistema operacional, mas no geral o que acontece é o seguinte:

1. O sistema lê a estrutura de dados superbloco, que está localizada no *bytes* 1024 do sistema de arquivos e o processa afim de conseguir o tamanho de cada bloco (1024 *bytes*). Cada grupo de blocos tem um total de 8192 blocos e 2016 *i-nodes*. Não há blocos reservados antes do início do primeiro grupo de blocos.

2. Em seguida o sistema lê os blocos 2 e 3 do sistema de arquivos, que correspondem a tabela de descritores de grupos do sistema de arquivos. Com isso ele descobre o *layout* dos grupos de blocos.

3. Como precisamos localizar o diretório "`dir1`"no diretório raiz, então o sistema processa o *i-node* 2 (*i-node* do diretório raiz). Usando o número de *i-nodes* por grupo , determinamos que *i-node* 2 está no grupo 0. Então o sistema usa a entrada da tabela de descritor de grupo para achar o bloco correspondente a tabela de *i-nodes* do grupo 0. Assim ele descobre que ela está no bloco 6.

4. Ele faz a leitura da tabela de *i-node* que está no bloco 6 e processa a segunda entrada. O *i-node* do diretório raiz mostra que seu conteúdo, ou seja, as entradas de diretório, estão localizadas no bloco 258 do disco.

5. Realiza-se então a leitura do conteúdo do diretório raiz, que está no bloco 258, e processa o conteúdo como uma lista de entradas de diretórios. As duas primeiras entradas são para '.' e '..'. O SO avança para a nova entrada afim de encontrar a de nome "`dir1`". Ao encontrá-la descobre que o *i-node* correspondente a ela é o de número 5033. O tempo de acesso do diretório raiz é atualizado.

6. É necessário agora encontrar onde *i-node* 5033 está localizado. Ao se dividir esse número pelo número de *i-nodes* por grupo percebemos que ele se encontra no grupo de bloco de número 2. Usando a entrada descritor de grupo do grupo 2, determinamos que sua tabela de *i-node* começa no bloco 16390.

7. Realiza-se a leitura da tabela de *i-nodes* do bloco 16.390 e processa a entrada 1001, que é o localização relativa do *i-node* 5033. O conteúdo do *i-node* revela que o conteúdo de "`dir1`"esta localizado no bloco 18431.

8. O sistema lê o conteúdo do bloco 18431 e o processa como uma lista de entradas de diretório a procura daquela correspondente ao arquivo "file1.321". Ao encontrá-la descobrimos que ela alocou o *i-node* 5110. Essa entrada de diretório é desalocada adicionando o seu comprimento ao mesmo campo da entrada anterior. Com isso os tempos de modificação dos metadados, de acesso ao arquivo e o de alteração

do conteúdo do diretório são atualizados e seus respectivos blocos aparecem nas entradas do arquivo *journal*.

9. Desse modo o sistema operacional segue o processo de exclusão tratando o *i-node* do arquivo. Ele processa o conteúdo do *i-node* 5110 que está no grupo 2 e decrementa em uma unidade o número de *links* relativo ao arquivo. Se o número de *links* chegar a 0 então o *i-node* deve ser desalocado. Para isso o *bit* correspondente deve ser alterado para 0, e o contador de *i-nodes* livres do superbloco e do descritor de grupo devem ser incrementados. Esses blocos que sofreram alteração também são registrados no arquivo *journal*.

10. Por fim é necessário liberar os blocos onde estão o conteúdo do arquivo em disco. O valor do *bit* correspondente de cada bloco é alterado para 0 no mapa de *bits* de blocos. O ponteiro de cada bloco do *i-node* é limpo e o tamanho do arquivo é reduzido no instante em que cada bloco é liberado, podendo atingir o valor de 0. Assim os tempo de modificação dos metadados, de acesso ao arquivo e de alteração do seu conteúdo são atualizados. O contador de blocos livres do descritor de grupos e do superbloco são atualizados e as mudança refletem entradas no arquivo *journal*.

3.1.5 Recuperação de Arquivos

Os algoritmos de alocação dos sistemas ExtX podem ajudar na recuperação do conteúdo de arquivos por existir uma tendência de reservar blocos próximos (geralmente no mesmo grupo de blocos) para um único arquivo. Dessa forma as buscas pelo conteúdo desses arquivos podem ser restringidas a uma parte do disco, ao invés de se fazer uma pesquisa mais abrangente.

Os sistemas que possuem estratégias de alocação de arquivos baseados no UFS (*Unix File System*) podem manter os blocos de *i-nodes* sem serem sobrescritos, guardando informações que ajudam em processos de recuperação. Isso acontece porque neles costuma-se buscar o primeiro *i-node* disponível para fazer novas alocações. Logo, diretórios que possuem as atividades de criação e exclusão de arquivos reduzidas, podem manter *i-nodes* por muito tempo sem serem sobrescritos. Em alguns casos a análise de metadados seria suficiente para realizar a recuperação dos arquivos.

Sendo mais específico, nos sistemas Ext2 os valores dos *i-nodes* não costumam ser apagados quando um arquivo é excluído. Então, por consequência os ponteiros para os blocos de dados ainda existirão. O elo existente entre as entradas de diretórios e os *i-nodes* são destruídas, mas é possível encontrar os *i-nodes* sem esse elo, usando uma busca por *i-nodes* desalocados. Existe a possibilidade dos ponteiros para os blocos indiretos terem sido realocado, o que torna mais difícil a recuperação de parte do arquivo.

Nos sistemas Ext3, os ponteiros das entradas de diretório e para os *i-nodes* são mantidos, porém os ponteiros para os blocos de conteúdo do arquivo são destruídos. Para se recuperar dados nesse sistema será necessário usar técnicas de *data carving*. A maioria dos sistemas irão alocar blocos de dados no mesmo grupo do *i-node* do arquivo o que reduz a dificuldade do procedimento mas não o torna de maneira alguma fácil. De modo geral, a área de busca pelo arquivo é reduzida. De qualquer maneira, pode acontecer de um grupo superar sua capacidade de armazenamento e o sistema tenha de alocar blocos de outros

grupos para o arquivo. A depender do arquivo que se esteja procurando, e da situação problema encontrada, deve-se considerar a possibilidade de busca em todo o sistema.

No Ext3 pode ainda ocorrer de existir uma cópia do *i-node* que acabou de ser excluído no arquivo *journal*. Nessa cópia pode existir todos os endereços de blocos de dados do arquivo. Bastando apenas processá-lo, desde que esse bloco não tenha sido sobrescrito (em nova alocação de arquivos), para que haja a recuperação.

Capítulo 4

Recuperação de Dados

A recuperação de dados é um processo usado para restaurar informações perdidas de mídia de armazenamento. Essa perda pode ser consequência de uma simples exclusão de arquivos ou pelo fato dessas mídias terem sido corrompidas, danificadas ou que não podem ser acessadas por vias normais usando o sistema operacional. Entenda a palavra mídias como: os discos internos ou externos de computadores, unidades de estado sólido (SSD), unidade de armazenamento USB, fitas, CDs, DVDs, RAID ou de outros aparelhos eletrônicos.

As áreas que mais fazem uso da recuperação de dados é a espionagem e a forense digital. Trata-se de um processo minucioso e tem por embasamento as abstrações usadas pelos sistemas operacionais.

Como vimos anteriormente, os computadores nada mais são que componentes físicos que usam a energia elétrica para produzir e armazenar informações. Usa-se 0's e 1's para tratar de todos os tipos de dados. Obviamente, não saberíamos lidar com esses tipos de informações se não houvessem as abstrações. Para que todos esses componentes trabalhem e gere algo útil para os humanos, nós criamos máscaras em cima de máscaras, camadas e mais camadas para que se veja ícones, diretórios, programas, jogos e muito mais nos monitores mundo a fora.

De acordo com [5], essas camadas que criam ilusões limitam o quanto podemos confiar a respeito das informações armazenadas em um computador sobre um sistema de arquivos. Apenas o nível físico, onde encontramos os domínios magnéticos dos discos, é real. Quanto mais alto o nível de abstração em que se estiver, maior a possibilidade de cometer erros ao interpretar as informações de um sistema.

Quando tratamos da destruição ou da recuperação de dados, novamente essas camadas de abstrações podem limitar as ações que realmente tem efeito sobre os discos. Elas podem nos dar visões do sistema que não correspondem ao que realmente encontramos ao analisá-los.

Podemos dizer portanto que excluir arquivos de um sistema é relativamente fácil, mas não é suficiente para destruir seus atributos muito menos o conteúdo em nível físico deles. Isso torna possível realizar procedimentos que permitam a recuperação desses arquivos que apesar de aparentarem indisponíveis para os usuários, ainda continuam nos domínios magnéticos dos discos.

A forense computacional usa desses conhecimentos para encontrar evidências criminais em seus casos de estudo. Apesar de não ser algo trivial muitos são os caso que são

solucionados com o uso de técnicas de recuperação de dados nesse ramo da ciência.

Olhando para o outro extremo, identificamos comportamentos um tanto triste, que requer atenção e cuidado. Existem casos em que proteger informações é essencial, sendo que, o seu descarte deve passar por vários procedimentos evitando as vezes perdas de vidas ou prejuízos incalculáveis. Uma multinacional ou uma pequena empresa pode querer destruir dados de um projeto finalizado que revelem informações importantes sobre a empresa e vir a sofre um processo de espionagem tendo seus direitos violados. Ou mesmo, listas ligadas a de processos judiciais com informações contidas a respeito de programas de proteção a testemunhas.

Tudo isso é possível ser feito por meio de procedimentos de recuperação de dados. A seguir detalhamos a respeito as partes do sistema EXT3 dando ênfase em onde obter informações para tentar recuperar os dados remanescentes em uma mídia de armazenamento. Seção 4.1 técnicas de análise, 4.2 Análise por categorias, 4.3 Técnicas de pesquisa em nível de aplicação, 4.4 resumo das estruturas e 4.5 a conclusão.

4.1 Os Dados

A motivação para a existência de sistemas de arquivos é muito simples: computadores precisão de métodos para armazenar, restaurar e organizar dados. O sistema de arquivos é o responsável por isso. Ele organiza os dados de maneira que os computadores consigam encontrá-los quando for preciso.

Como vimos em seções anteriores, todo sistema de arquivos precisa de uma organização. Em cada partição de um disco podemos separar os dados em categorias de acordo com a funcionalidade de suas informações. Assim temos os dados usados na administração do sistema de arquivos, os dados que caracterizam os arquivos (metadados), os dados que ajudam na interação do ser humano com o computador (diretórios), os dados dos conteúdos de cada arquivo e entre outros com suas próprias funcionalidades.

Os que tratam da administração do sistema contém informações gerais sobre o sistema. Eles definem e constroem uma estrutura, como se fosse um mapa dos discos onde descreve suas áreas e características: como , por exemplo, o local onde as estruturas estão armazenadas, qual o tamanho dos blocos do discos, quantidades de blocos e assim por diante.

Os metadados guardam informações a respeito dos próprios dados. Esses são dados que descrevem outros dados. Exemplos desse tipo de estrutura são as entradas de diretórios das tabelas FAT, a tabela mestra de arquivos do NTFS e as estruturas de *i-node* dos sistemas baseandos no UFS (*Unix File System*) e EXT3.

Os dados de conteúdo são os que realmente armazenam o conteúdo dos arquivos. A maioria dos dados de um sistema de arquivos pertencem a essa categoria e normalmente são armazenados em conjuntos que formam *containers* com tamanhos pré-definidos. Cada sistema operacional dá um nome a esses *containers*. No caso do Windows são chamados de *clusters*, já no Unix, de blocos.

A categoria de dados de interface humana contém dados a respeito do nome de cada arquivo. Na maioria dos sistemas de arquivos esses dados estão localizados no conteúdo dos diretórios.

4.1.1 Dados Essenciais e não Essenciais

Como é possível perceber, existem dados que são necessários para se realizar a leitura ou escrita de um arquivo - como os ponteiros para o conteúdo de um arquivo, o nome de um arquivo e o ponteiro para as estruturas de metadados - bem como existem outros, que existem por conveniência mas que não são usados para as funcionalidades básicas do sistema de arquivos. Estes dados são caracterizados como não essenciais e aqueles como essenciais.

O motivo de haver essa diferenciação está no fato de que devemos confiar nos dados essenciais, ao contrário do que ocorre com os outros. Todo endereço de conteúdo de um arquivo tem um valor que aponta para onde os dados do arquivo está armazenado. Esse valor é essencial porque se ele for falso, o usuário não seria capaz de ler o arquivo. Por outro lado, conseguimos recuperar um arquivo mesmo que o tempo de criação deste não seja o verdadeiro.

Claro que os sistemas operacionais podem de alguma forma exigir que certos valores estejam corretos, de forma que impeça a abertura de um arquivo em caso contrário. Mas ainda assim isso não quer dizer que esses valores sejam essenciais. Isso deixa claro que cada sistema operacional pode implementar procedimentos específicos para operar com o sistema de arquivos. Muitos podem, por exemplo, implementar um sistema de arquivo FAT e cada um pode realizar a exclusão de arquivos usando técnicas diferentes. Principalmente porque cada sistema tem seu propósito de existir, podendo alguns primar mais pelo desempenho e outros mais pela privacidade no trato das informações.

Assim, quando estamos interessados na recuperação de arquivos, não é suficiente perguntar a qual sistema de arquivos o alvo a ser recuperado pertencia. E sim, "como recuperar um arquivo de determinado sistema de arquivo armazenado e em determinado sistema operacional?".

4.2 Análise por Categorias

Conforme [1], quando procuramos dados em um disco, devemos fazê-lo com base em suas propriedades e questionando onde ele poderia aparecer no disco. Baseado na ideia de onde esperamos que o dado esteja, podemos identificar a categoria de dados apropriada e buscar a respectiva técnica de análise para tentar recuperá-lo. Por exemplo, se procuramos um arquivo de imagem com a extensão "JPG", seria importante utilizar análises com foco na categoria de dados dos diretórios já que ela guarda informações a respeito do nome dos arquivos e suas extensões. Se por outro lado, estamos procurando um arquivo que possui determinadas características, tentar-se-ia usar análises dos blocos de metadados e por ai vai.

4.2.1 Categoria Arquivos de Sistema

Essa categoria contém dados gerais que identifica um sistema de arquivo e permite saber onde outras estruturas importante do sistema estão localizadas. Análise de dados nessa categoria é necessária para todos os tipos de análise de sistemas de arquivos porque é nela que se descobre onde estão localizadas as outras partes do sistema de arquivo.

Devido a importância dessa categoria de dados, é comum haver cópias de segurança deles em outros lugares no disco. Caso algum dado esteja corrompido ou perdido, é possível localizar áreas de backup de seu conteúdo.

Os dados dessa categoria são tipicamente valores independentes os quais não se pode fazer muito além de mostrá-los. Se for interessante saber qual sistema gerou o volume, ou informações de *layout* de disco, tamanhos de blocos e locais de armazenamento propícios a esconder dados, é daqui que se obtém essas informações.

Uma das ferramentas do *"The Sleuth Kit and Autopsy - TSK"*, chamada "fsstat", é capaz de mostrar dados dessa categoria de arquivos. Outra bastante utilizada é o `dumpe2fs`.

4.2.2 Categoria Conteúdo

Os dados que pertencem a essa categoria se localizam na parte do disco responsável por armazenar os conteúdos dos arquivos e dos diretórios. Os dados são organizados em unidades de tamanhos iguais (blocos). Essas unidades de armazenamento podem ser encontradas em dois estados: alocadas ou não alocadas. Todo sistema de arquivos mantém estruturas que controlam o estado dessas unidades.

A depender da estratégia de alocação de unidades de armazenamentos em um sistema operacional, técnicas diferentes podem ser usadas para se tentar recuperar arquivos. No geral, quando um arquivo é excluído, sua unidade de dados é liberada para que novos arquivos possam usá-las. A maioria dos SOs não destroem o conteúdo dessas unidades após serem desalocadas. Com os dados em disco, apesar de não acessíveis diretamente via SO, esses dados podem ser recuperados.

A análise de dados dessa categoria deve ser direcionada para regiões de dados alocados e não alocados, com pesquisas em baixo nível de abstração e muitas vezes com identificação de padrões entre os blocos. Isto é necessário para tentar identificar, como se fosse um quebra-cabeça, se um bloco poderia ou não ser continuação de outro. Por exemplo, suponha que tenhamos um bloco cujo o conteúdo são textos em ASCII. Suponha também que encontremos algum tipo de endereçamento em sequência dele com outro bloco. Caso esse novo bloco possua valores binários não legíveis, saberíamos que provavelmente não pertencem ao mesmo arquivo. Apesar da ligação entre os endereços.

Busca em Unidades de Dados

A busca em unidades de dados é uma técnica usada quando o perito computacional sabe o endereço onde o arquivo possa estar no disco. A teoria por trás desse tipo de análise é relativamente simples. O investigador informa à ferramenta o endereço e esta retorna os *bytes* da unidade de dados. O investigador analiza seu conteúdo e os reúne em um único arquivo recuperando-o.

Existem diversos leitores hexadecimais e ferramentas capazes de realizar esse trabalho. Uma delas é o `dcat` do *TSK*. Para aqueles com maior conhecimento pode ser usado o comando `dd` do linux.

Pesquisa em Nível Lógico

Nesta técnica, sabemos o conteúdo dos dados que procuramos mas não sabemos onde eles poderiam está nos discos. Ela costuma ser conhecida como pesquisa em nível físico, pois a busca ocorre nos setores físicos dos discos. A pesquisa em nível lógico no sistema de arquivo procura em cada unidade de dados por uma frase específica ou algum valor conhecido. O problema é que nem sempre os arquivos estão alocados em unidades consecutivas de dados. Se um arquivo que estamos procurando estiver fragmentado em unidades não consecutivas, esse tipo de pesquisa não retornará bons resultados.

Basta pensarmos analogamente em um jogo de futebol, onde desejamos procurar por uma família especifica no estadio. A pesquisa em nível lógico iniciaria a busca na primeira fila da arquibancada e examinaria grupos de quatro em quatro pessoas. Se a família não estiver em lugares consecutivos não conseguiremos encontrá-la. A pesquisa poderia ser alterada para apenas uma única pessoa da família, porém é provável que encontremos muitas pessoas que se pareçam com a que estamos procurando, mas que não seja aquela que desejávamos localizar.

Estado de Alocação da Unidade de Dados

Algumas ferramentas de recuperação de dados são capazes de extrair dos discos todas as unidades de dados que não estão alocadas. Caso o investigador saiba que os dados que procuram estão nesses blocos de discos, pode-se criar um arquivo separado com uma imagem dessas unidades de dados do sistema de arquivos.

A desvantagem é que, após a extração dos dados para um único arquivo, não se pode manipular os arquivos em software algum, já que o resultado é uma coleção de *bits* sem nenhuma estrutura de arquivo.

Na ferramenta *TSK*, o programa `dls` é capaz de extrair todos os blocos de dados não alocados para um arquivo e o programa `dcalc` busca, a partir dos dados encontrado pela primeira ferramenta, onde esses dados estão no sistema de arquivos original.

4.2.3 Categoria de Metadados

Como falamos anteriormente, nessa categoria de dados encontramos as descrições dos arquivos do sistema. A maioria da estruturas de metadados são armazenadas em uma parte fixa do disco ou em uma tabela de tamanho variável onde cada entrada possui endereços dos dados da categoria de conteúdo.

Quando o arquivo é removido, a entrada de metadados é marcada como disponível para nova alocação, sendo que as ferramentas de limpeza segura de disco podem remover certos valores.

A análise desses dados permite filtrar melhor os arquivos os quais estamos procurando para realizar a recuperação. Existem dois principais métodos para se recuperar arquivos. Um deles é baseado nos metadados e o outro é baseado no *journal* - discutiremos mais a diante. Quando os metadados de um arquivo excluído ainda persiste na estrutura, usa-se o primeiro método de recuperação. Quando a estrutura de metadados foi realocada para outro arquivo ou quando esta foi limpa por algum programa ou sistema operacional, usa-se o outro.

Depois de encontrar a estrutura de metadados que descreve um arquivo no sistema, fica fácil recuperar os dados por ela apontados. É o mesmo que ler o conteúdo de um arquivo alocado, uma vez que os endereços dos blocos de conteúdo do disco ainda existem. Ao mesmo tempo que parece trivial, deve-se tomar certos cuidados pois os blocos de conteúdo podem não estar sincronizadas com os endereços da estrutura de metadados. Ou seja, eles podem ter sido realocados para um outro arquivo e este, por sua vez, pode também já ter sido removido do sistema. Além do mais, saber se a unidade de dados, que a estrutura de metadados aponta, foi realocada ou não pode ser também uma tarefa difícil.

Assim, nas tentativas de recuperação de dados os investigadores podem usar de informações a respeito dos dados, que se deseja recuperar, facilitando as buscas. Se sabemos que o dado foi removido recentemente, é provavel que tanto sua estrutura de metadados como os blocos de conteúdo ainda estejam preservados. Buscas em estruturas não alocadas podem gerar melhores resultados que buscar no disco inteiro.

4.2.4 Categoria dados de Interface Humana

A categoria de dados de interface humana permite que os arquivos seja referenciados pelo seu nome ao invés de o serem pelo endereço de seu metadado. Dessa categoria de dados estão incluídos os nomes dos arquivos e o endereço de metadados. Alguns sistemas de arquivos inserem informações a respeito do tipo de arquivo e dados efêmeros, mas não é o padrão. Estamos nos referindo aos diretórios.

Recuperação Baseada no Nome do Arquivo

Os arquivos removidos podem ser recuperados usando seus metadados. Para conseguir acesso às estruturas destes e posteriormente realizar a recuperação usamos essa técnica. Ela consiste em identificar nomes de arquivos deletados e seus respectivos endereços de metadados. Após identificá-los o trabalho de recuperação é feito pela técnica de recuperação explicada na categoria de metadados.

Em [1], o autor exemplifica a técnica. Considere um arquivo que possui o nome file1.dat e um ponteiro para os atributos do arquivo com valor 100 4.1 (A). Esse arquivo foi removido e tanto o nome do arquivo quanto sua estrutura de metadados deixaram de estar alocados. Porém nem o ponteiro do diretório nem o nome do arquivo foi destruido. Um novo arquivo com o nome file2.dat foi criado em outra estrutura de interface humana, apontando para a estrutura de metadados anterior 4.1(B). Posteriormente, esse arquivo foi também excluído e suas estruturas se tornaram não alocadas 4.1(C). Se chegarmos a este estado em uma tentativa de recuperação de um arquivo teríamos duas opções mas não saberíamos a que arquivo o endereço de número 100 leva. Isso pode piorar se tivermos um novo arquivo o qual possui outro endereço de estrutura de metadado e que deseja alocar a unidade de conteúdo 1000.

As análise de dados dessa categoria estão baseadas no que ela normalmente tem como responsabilidade. Isso significa que ela terá de listar os nomes dos arquivos e diretórios do sistema de arquivos. Uma outra ideia seria listar as extensões dos arquivos com a intensão de agrupá-los por tipos de arquivo. Muitos sistemas não se preocupam em realizar uma limpeza do nome dos arquivos que foram excluídos, o que facilita a vida dos profissionais que recuperam arquivos.

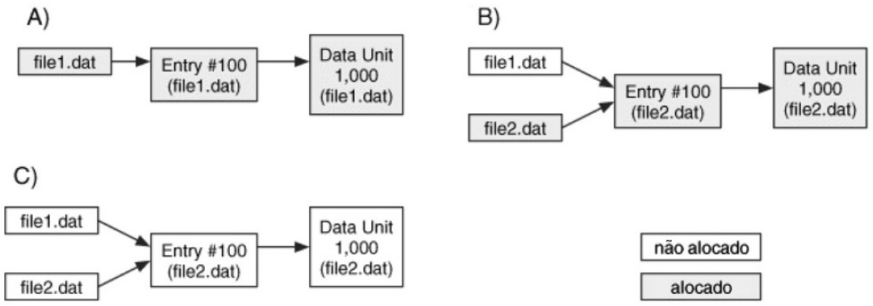

Figura 4.1: A sequência de estados depois da alocação de arquivo e sua remoção. Em (C) existem dois arquivos removidos que apontam para a mesma estrutura de metadados e não é possível saber a quem o conteúdo pertence. Adaptado de [1].

Como já falamos a ideia para obter a listagem desses nomes está em localizar o diretório raiz do sistema de arquivo. Os dados do diretório raiz está armazenado em uma estrutura de metadados. A partir dessa estrutura localizamos os blocos de conteúdo dele. Temos então o nome dos arquivos e dos diretórios dentro do raiz e os ponteiros para as estruturas de dados do mesmo. Para cada diretório seguinte da árvore de diretórios, basta repetir os mesmos passo e averiguar os arquivos neles armazenados como mostra a figura 4.2.

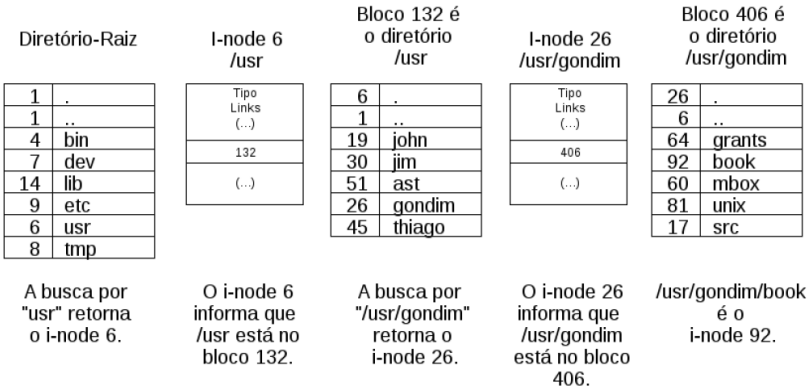

Figura 4.2: Esquema de busca de arquivos em diretórios de um sistema de arquivos.

4.2.5 Categoria de Aplicação

Alguns sistemas de arquivos contêm dados que pertencem à categoria da aplicação. Estes dados não são essenciais para o sistema de arquivos, e normalmente só existem porque tornam o sistema de arquivo mais eficiente. Tecnicamente, qualquer arquivo que um sistema operacional ou um aplicativo cria poderia ser compreendido como um recurso a mais de um sistema de arquivos. Esta seção aborda um dos recursos mais comuns dos dados da categoria de aplicação, que é chamado de *journaling*.

Journals de Sistemas de Arquivos

Como qualquer usuário de computador sabe, não é raro acontecer de um computador travar ou desligar de forma inesperada. Se nesses instantes o sistema operacional estivesse realizando alguma gravação de dados em disco ou se ele houvesse dados esperando para serem gravados, o sistema de arquivos pode ficar em um estado inconsistente. Um exemplo poderia ser uma estrutura de metadados que está alocada juntamente com suas respectivas unidades de dados, mas com os ponteiros que os ligam corrompidos.

Para mapear as inconsistências, o sistema operacional costuma executar programas que verificam o sistema de arquivos e buscam por ponteiros perdidos e outros sinais de corrupção. Isso pode levar um tempo razoável principalmente se os sistemas de arquivos são grandes. Para tornar o trabalho desses programas mais leve, alguns sistemas de arquivos implementam os "*journals*". Antes de quaisquer alterações nos metadados do sistema de arquivos, é criada uma entrada num arquivo *journal* que descreve as mudanças que irão ocorrer. Depois de feitas as alterações, uma outra entrada é criada no mesmo arquivo para registrar as mudanças efetivadas. Se o sistema falhar, os programas que buscam por falhas lêem o *journal* e toma conhecimento das operações que não foram concluídas. O programa então ou completam as mudanças ou as desfazem colocando o arquivo de volta ao estado original.

Muitos sistemas de arquivos agora suportam *journaling* porque economiza tempo durante a inicialização de sistemas de grande porte. Os arquivos *journals* pertencem a categoria de aplicação, pois não são necessários para que o sistema de arquivos possa operar. Eles existem apenas para fazer a verificação de consistência dos dados de forma mais rápida.

Esses arquivos podem ser úteis para se entender o estados de certos arquivo ou tentar recuperá-los. O registro desses arquivos podem ajudar a entender o que aconteceu com o sistema ao realizar a reconstrução dos eventos registrados. A maioria das ferramentas de recuperação de dados não processam o conteúdo de um arquivo *journal* mas algumas podem listar seu conteúdo como o `jls` e o `lcat` do kit *TSK*.

4.3 Outras Técnicas

Nessa seção iremos abordar um pouco a camada de aplicação e discutir algumas técnicas que podem ser usadas para recuperar arquivos apagados e organizá-los para análise. Ambas estas técnicas dessa seção usam o fato de que muitos arquivos possuem estruturas fixas e que possuem valores que funcionam como assinatura. Essa assinatura pode ser utilizada para determinar o tipo de um arquivo. O comando file presente em muitos sistemas UNIX possui um banco de dados de assinaturas que ele usa para identificar a estrutura de um arquivo desconhecido (`ftp://ftp.astron.com/pub/file/`).

4.3.1 *Data Carving*

Data Carving ou *File Carving* é um processo de busca em que um bloco de dados é analisado a procura de assinaturas que correspondem a tipos de arquivos conhecidos. O resultado do processamento é uma coleção de arquivos que contém as assinaturas da busca. Isto é normalmente realizado em espaços não alocados de um sistema de arquivos

e permite recuperar arquivos que não estão ligados a estruturas de metadados alguma. Por exemplo, uma imagem JPEG tem um cabeçalho padrão e valores que identificam seu fim. Para recuperar um arquivo JPEG a técnica de *carving* busca no espaço não alocado por meio de uma ferramenta, extrair os dados entre o cabeçalho JPEG e sua marcação de fim. Caso não sejam encontradas as marcações o método os arquivos não são recuperados.

De acordo com Merola [9], quando usamos uma técnica básica de *file carving*, assumimos três hipóteses a respeito dos arquivos alvos:

1. O início deles, ou seja, seus cabeçalhos não estão sobre escritos.
2. Não estão fragmentados.
3. E não estão compactados ou criptografados.

Existem técnicas mais avançadas que fazem buscas por arquivos fragmentados onde até mesmo fragmentos não sequenciais, fora de ordem ou mesmo perdidos se consegue recuperar os arquivos. Nesses caso não há outra forma a não ser retornar às informações da estruturas internas dos arquivos, ainda assim com grande chances de estarem corrompidas.

Um exemplo de ferramenta que realiza essa técnica de recuperação de dado é a *Foremost* (encontrada no site `http://foremost.sourceforge.net`) desenvolvida pelos agentes Kris Kendall e Jesse Kornblum do Departamento de Investigações Especiais da Força Aérea dos Estados Unidos. Essa ferramenta analisa o sistema de arquivo na sua forma bruta ou uma imagem de disco com base em um arquivo de configuração que armazena uma lista de assinaturas validas. Cada assinatura contém um valor conhecido, o tamanho máximo do arquivo, se o valor do cabeçalho é sensível a letras maiúsculas e minúsculas, a extensão mais comum do tipo de arquivo, e um valor que indica fim de arquivo como campo opcional.

Outra ferramenta similar a essa é o software Lazarus disponibilizado no *The Coroner's ToolKit - TCT* desenvolvido por Dan Farmer (`http://www.porcupine.org/forensics/tct.html`).

4.3.2 Classificação por Tipo de Arquivo

O tipo de arquivo também pode ser usado para organizar os arquivos em um sistema de arquivos. Se estivemos à procura de um tipo específico de dados, podemos classificar os arquivos com base em sua estrutura de conteúdo. Um exemplo de execução dessa técnica seria usar o comando `file` em cada arquivo e agrupar os tipos de arquivos semelhantes. Isso colocaria todas as imagens juntas bem como todos os executáveis juntos. Muitas ferramentas de análise forense tem esse recurso, mas nem sempre é claro se eles estão classificados com base na extensão do arquivo ou por meio da sua assinatura. A ferramenta `sorter` do TSK classifica os tipos de arquivos com base em sua assinatura.

4.4 Sistemas de Arquivos Específicos

A tabela 4.1 resume as estruturas e os dados de cada categoria para os sistemas de arquivos dos tipos EXTX e UFS.

FS	Arquivos de Sistema	Conteúdo	Metadado	Interface Humana	Aplicação
EXTX	Superblocos, Grupos, Descritores	Blocos e *Bitmaps*	*I-nodes*, *i-node bitmap* ,atributos estendidos	Entradas de Diretórios	Arquivos *Journals*
UFS	Superblocos, Grupos, Descritores	Blocos, fragmentos, *bitmap* dos fragmentos e *bitmaps*	*I-nodes*, *i-node bitmap* , atributos estendidos	Entradas de Diretórios	N/A

Tabela 4.1: As estruturas de dados por categorias para os sistemas de arquivos.

4.5 Considerações

Sistemas de arquivos Unix atuais não espalham o conteúdo de um arquivo de forma aleatória ao longo do disco. Em vez disso, eles são extraordinariamente bons em evitar a fragmentação de arquivos, mesmo depois de anos e anos de uso intenso. Porém, percebemos que as técnicas de recuperação pressupõem a existência dos dados no disco. Em cada uma delas, só é possível realizar a recuperação se os dados não foram alterados. É nesse caminho que propomos no próximo capítulo as opções de remoção segura do nosso trabalho.

Capítulo 5

Remoção Segura de Dados

A remoção segura de dados torna-se mais comum e muitos sistemas operacionais já colocam-na como um dos seus recursos padrão. Normalmente programas desenvolvidos por terceiros é que realizam a limpeza nos discos. Eles não costumam ser partes do kernel do sistema operacional. Com isso tornam-se refém destes na hora de executar algumas tarefas e podem não ser tão efetivos no seu propósito. Por exemplo, se considerarmos o caso hipotético de existir um sistema operacional que simplesmente não implemente a escrita de zeros consecutivos em disco (realize ao invés disso uma marcação especial), uma ferramenta que mande ele escrever zeros sobre determinados dados, pode não sobrescrever dados importantes que deveriam desaparecer, ou seja, continuariam disponíveis para futuras tentativas de recuperação de dados.

Neste capítulo veremos (5.1) detalhes da ferramenta FRS criada nesse trabalho para realizar limpeza no sistema de arquivos EXT3 usando uma abordagem por categoria de dados, conforme o analizamos no capítulo anterior. Veremos também quais foram os procedimentos e os resultados que obtivemos no experimento (5.2) realizado.

5.1 Limpeza no Sistema de Arquivos

Nesse trabalho focamos nossos estudos para o sistema de arquivo EXT3. Temos interesse em analisar o efeito que tem, algumas modificações no processo de exclusão de arquivos, sobre a recuperação de dados.

A ferramenta FRS possui 6 níveis de operação. Em cada um deles ela atua em um conjunto de estruturas do sistema de arquivos sobrescrevendo ou alterando dados importantes para sua recuperação. Ela foi escrita na linguagem C direcionada para executar no OpenSuse 12.3. Sua execução em outros sistemas podem apresentar situações inesperadas.

Por manipular arquivos maiores que 2GB, ela utiliza a função `seek64()`. Isso gera a necessidade de execução do comando `# getconf LFS_LDFLAGS` em sistemas de 32bits antes que entre em funcionamento.

No seu primeiro nível, o que a ferramenta faz é decrementar o número de entradas de diretório que apontam para o *i-node* do arquivo. Realizamos isso usando a função `unlink()`. No seu segundo nível, ela acrescenta como parte do processo de exclusão, a alteração do nome do arquivo. No terceiro, incorpora a sobrescrita de *i-nodes* e a limpeza do *journal*. No nível seguinte agrega a sobrescrita do primeiro bloco do arquivo com zeros. No quinto nível, a ferramenta realiza o mesmo que o terceiro acrescido de

sobrescrita com valores aleatórios dos três primeiros blocos de dados. E no último, há também a sobrescrita de todo o arquivo.

A tabela a seguir 5.1 resume o comportamento da ferramenta FRS em cada um dos seus níveis.

Nível de Destruição	**Operações**				
	Unlink	***Wipe* Nome**	***Wipe* *i-node***	***Wipe* *Journal***	***Wipe* Blocos de Dados**
# 1	✔				
# 2	✔	✔			
# 3	✔	✔	✔	✔	
# 4	✔	✔	✔	✔	1º bloco
# 5	✔	✔	✔	✔	1º,2º e 3º blocos
# 9	✔	✔	✔	✔	Todo o arquivo

Tabela 5.1: Níveis de destruição de dados da ferramenta FRS.

5.1.1 Lista de Diretórios

Neste passo tentamos retirar o vínculo existente entre o nome do arquivo e a estrutura de metadados vinculada a ele. Assim o ideal seria destruir, antes da remoção do arquivo, seu nome e ponteiros para outras estruturas.

Uma das técnicas usadas é sobrescrever os valores dos campos nome e *i-node* das entradas de diretório, de forma que apesar das análises de recuperação indicarem a existência de um arquivo, ela não seja capaz de recuperá-lo usando seu nome. Por exemplo, poderia substituir o nome do arquivo `passwords.txt` por `abcdefg.321`. O problema que podemos enfrentar é a forma como o sistema operacional realiza algumas operações. Por exemplo, se ao invés de sobrescrever a entrada de diretório correspondente ao arquivo, o sistema operacional simplesmente alocar uma nova entrada com o novo nome e liberar a anterior sem modificá-la, a alteração não terá o efeito desejado já que a informação permaneceria no disco.

Brian Carrie [1] defende que reorganizar a lista de nomes de modo que uma das entradas da lista de nomes sobrescreva a entrada alvo (da mesma maneira que é implementado pelo NTFS) seria uma das maneira mais interessantes de realizar essa operação. Apesar de mais complexo esse método aparenta ser mais efetivo pois o investigador, que deseje recuperar um arquivo pelo nome, nem mesmo saberia que o arquivo esteve naquele diretório. As entradas continuariam com valores reais, mas aquela, que guardava o nome do arquivo alvo, possuiria valores diferentes dos originais.

Na FRS resolvemos apenas por sobrescrever o nome do arquivo alvo usando a função rename. Para isso usamos o dispositivo `"/dev/random"` do *linux* como um gerador de valores aleatórios limitando a saída aos caracteres ASCII entre 48 e 123 para compor as letras do nome do arquivo como percebemos no código a seguir. Sabendo o comprimento do nome do arquivo, é possível sobrescrever todo o nome original.

O dispositivo `"/dev/random"`, de acordo com [14], reúne a entropia de eventos que ocorrem no computador, como as entradas dos *drivers* de usuário ou interrupções do kernel para gerar dados aleatórios.

```
90 /* Arquivo:      */
91 for(i=0;i<length;i++){
92   do{
93     c = get_char("/dev/random");
94   }while(c<48||c>123);
95   string + i = c;
96 }
```

Outra opção seria usar um gerador de números pseudo-aleatórios, como o "Mersenne Twister"[4], e limitar a saída ao tamanho de uma string a qual é composta por todos os caracteres de texto. Assim poderíamos criar um novo nome juntando cada letra até completar o tamanho do nome original.

```
100 /* Arquivo:      */
101 long int c;
102
103 c = mersennetwister(srand(time()));
104
105 /* str length = 75 */
106 str[]={a,b,c,...,A,B,C,...,0,1,2,...};
107
108 return str[(c%75)];
```

As duas soluções se mostraram eficientes em gerar novos nomes aleatórios não onerando significativamente a performance da ferramenta. Abaixo na figura 5.1 vemos exemplos de saídas usando os dois métodos.

5.1.2 Estruturas de Metadados

Os campos de tamanho, tempo e principalmente os de endereços seriam campos importantes quando se tenta recuperar arquivos. A destruição destes poderia dificultar a recuperação de dados. O tamanho e os tempos passam tanto informações a respeito da criação de um arquivo, como da sua exclusão. Os campos que armazenam endereços de blocos identificam onde está o conteúdo do arquivo. É necessário portanto que essa informação também seja destruída. Para que isso ocorra basta sobrescrevê-los com zeros ou valores randômicos.

Na implementação dos sistemas EXT3, pelo menos no OpenSuse 12.3, já existe a preocupação por parte do sistema operacional em inserir zeros nos campos de endereços de conteúdo dos arquivos no momento de sua exclusão. Além disso, o campo referente ao tamanho também recebe o valor zero.

Abaixo percebemos os dados de um *i-node* de um arquivo antes de sua exclusão. A primeira imagem (5.2) nos trás uma visão de alto nível dos dados contidos no *i-node* 13 de um disco. A segunda (5.3) nos trás 512 *bytes* do disco, sendo os primeiros 256 correspondentes ao mesmo *i-node* no sistema de arquivos EXT3.

Em seguida mostramos esse *i-node* após a exclusão do arquivo vinculado a ele e depois de desmontar (comando `umount`) a sua unidade de disco. A primeira 5.4 foi gerada pela ferramenta TSK e a segunda 5.5 usando o comando `dd`.

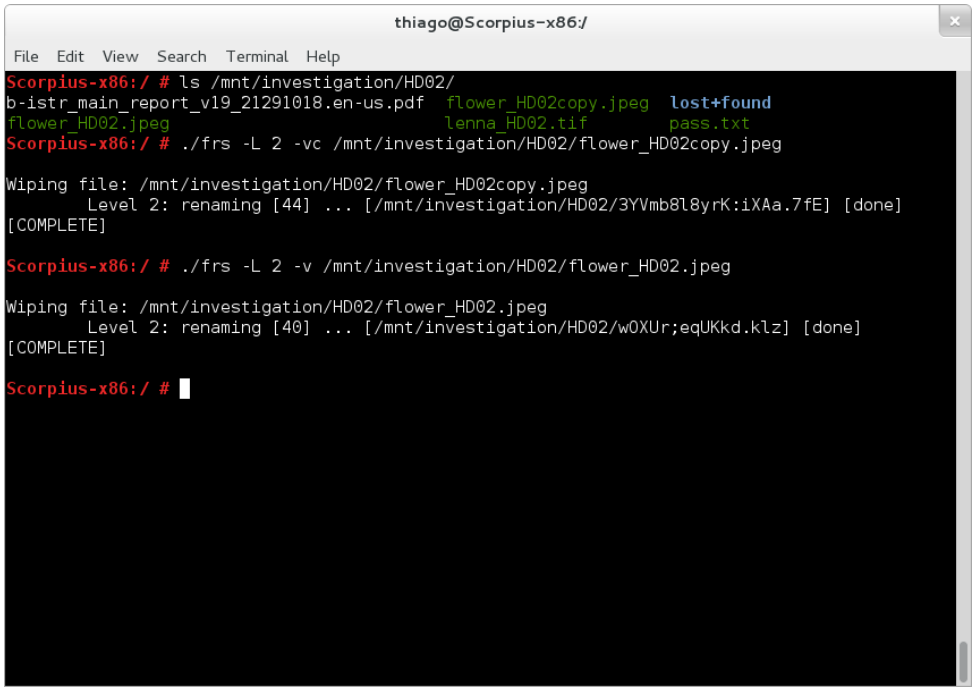

Figura 5.1: Diferença entre os algorítmos da FRS responsáveis pela alteração de nomes dos arquivos. O parametro -c indica a utilização do /dev/random.

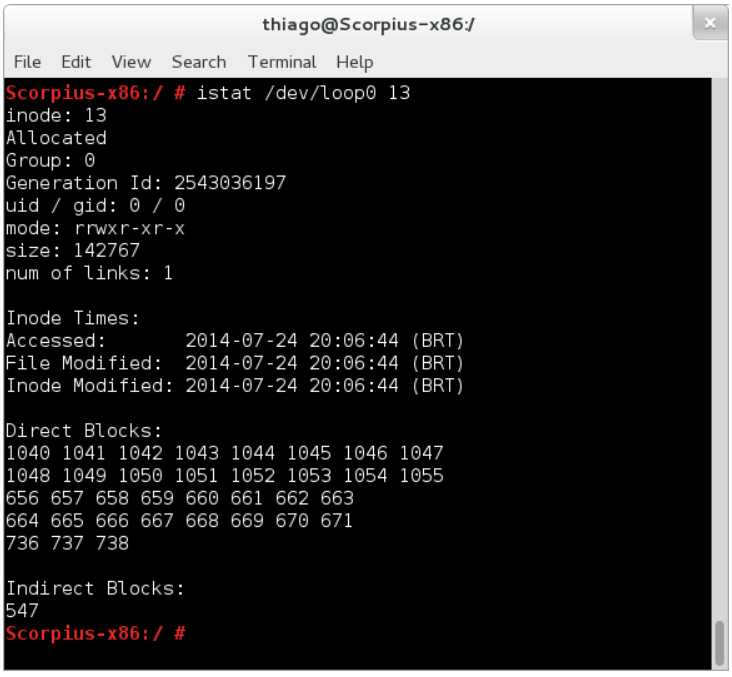

Figura 5.2: Ferramenta `istat` do TSK mostra o *i-node* antes da exclusão de um arquivo no EXT3.

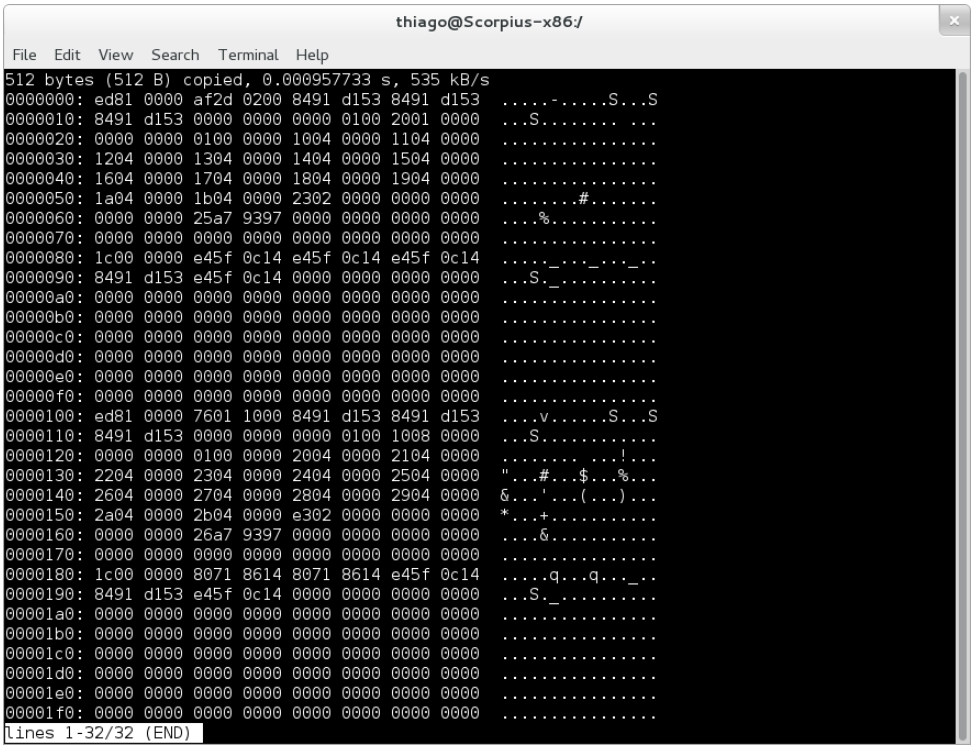

Figura 5.3: Ferramenta `dd` do linux mostra os *bytes* do *i-node* antes da exclusão de um arquivo no EXT3.

Observamos nessas imagens que os dados responsáveis pelos endereços de blocos de conteúdo tornaram-se zeros. Os tempos de modificação do arquivo e modificação do i-node foram atualizados da mesma maneira que aconteceu com os dados de tamanho e o números de links.

Quando a ferramenta FRS roda em um nível maior ou igual a 3 ela realiza a sobreposição de *i-nodes*. Isto é, sobrescreve os dados do *i-node* do arquivo a ser excluído pelos de outro do sistema (imediatamente anterior ou o posterior).

Isso substituiria os dados do *i-node* alvo da remoção pelos de outro antes que o processo do sistema operacional se realiza-se. Ilustramos esse procedimento por meio da figura 5.6 com um *i-node* de 256 *bytes*.

5.1.3 Conteúdos dos Arquivos

Nesta parte do sistema de arquivo a melhor estratégia é sobrescrever os dados dos arquivos. Usar zeros ou valores randômicos no lugar do disco onde se encontram os dados dos arquivos.

A ferramenta FRS antes de realmente excluir o arquivo, o abre e escreve 0's (se nível 4) ou valores aleatórios (se nível 5 ou 9) sobre parte ou sobre todo o conteúdo do arquivo usando os dispositivos `"/dev/zero"` ou `"/dev/random"` como fonte de dados.

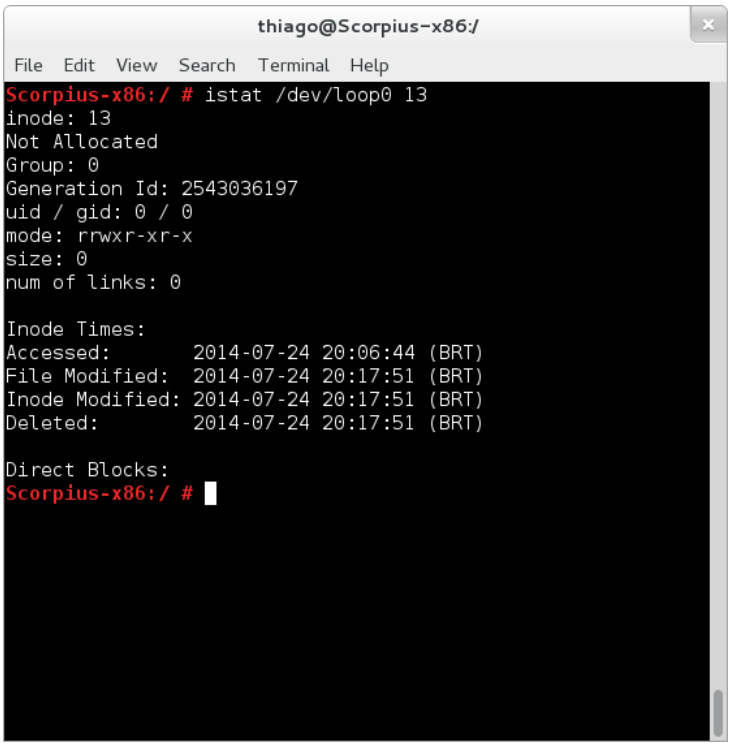

Figura 5.4: Ferramenta `istat` do TSK mostra o *i-node* após a exclusão de um arquivo no EXT3 em contra ponto à figura 5.2.

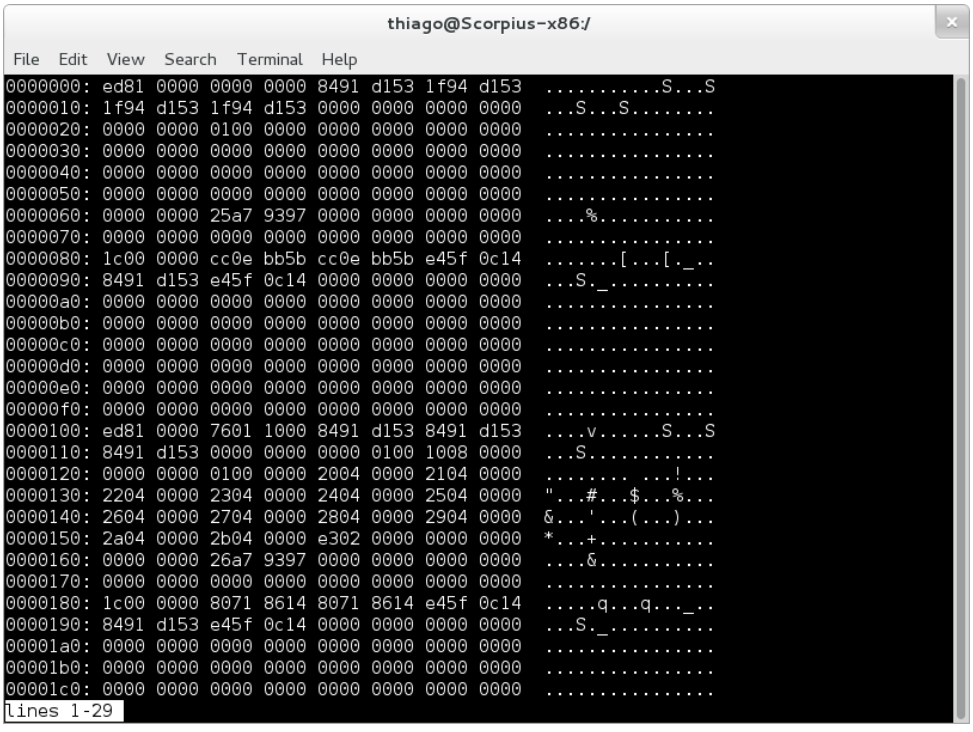

Figura 5.5: Ferramenta `dd` do linux mostra os *bytes* correspondentes ao *i-node* após a exclusão de um arquivo no EXT3. Um contra ponto à figura 5.3

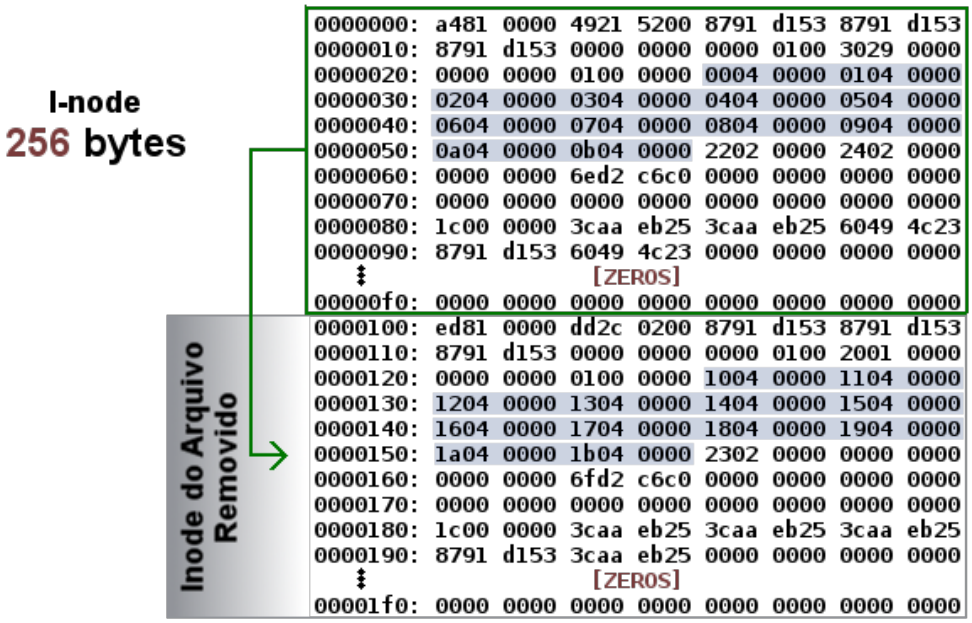

Figura 5.6: Ilustração da substituição feita pela FRS em um processo de sobrescrita de *i-node*.

5.1.4 *Journaling*

Por fim, temos o problema do *"journaling"*. Ele pode armazenar todo os metadados e , em alguns caso, até mesmo o conteúdo de um arquivo que foi excluído. Assim, mesmo que se destrua as informações constantes nas estrutura de metadados de um arquivo, é possível que elas existam em algumas das entradas do arquivo *"journal"*. Dessa forma é necessária a limpeza dos dados desse arquivo para que se dificulte a sua recuperação.

A ferramenta FRS, quando rodada nos níveis maiores do que ou iguais a 3 limpa com zeros as entradas do arquivo *"journal"*, evitando técnicas de recuperação de dados por meio deles.

5.2 O Experimento

Nas próximas seções descreveremos os experimentos realizados neste trabalho utilizando a FRS e imagens de disco.

5.2.1 Preparação

Iniciamos o processo criando seis imagens de discos limpas como zeros utilizando o comando `dd` do linux, conforme mostra a imagem 5.7 alterando o "x"do nome da imagem `hd0x.img` (`x = 1,2,3,4,5 e 9`) para se referir a cada uma delas.

Após a execução do comando surgiram as imagens de disco que aparecem na figura 5.8.

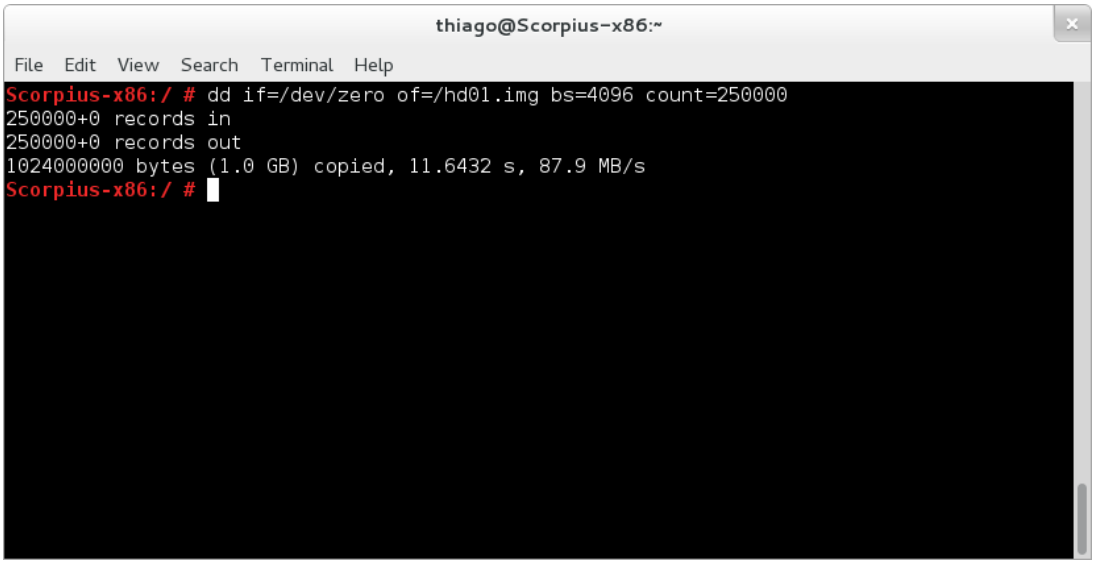

Figura 5.7: Comando dd criando uma imagem de disco com todos os seus campos zerados.

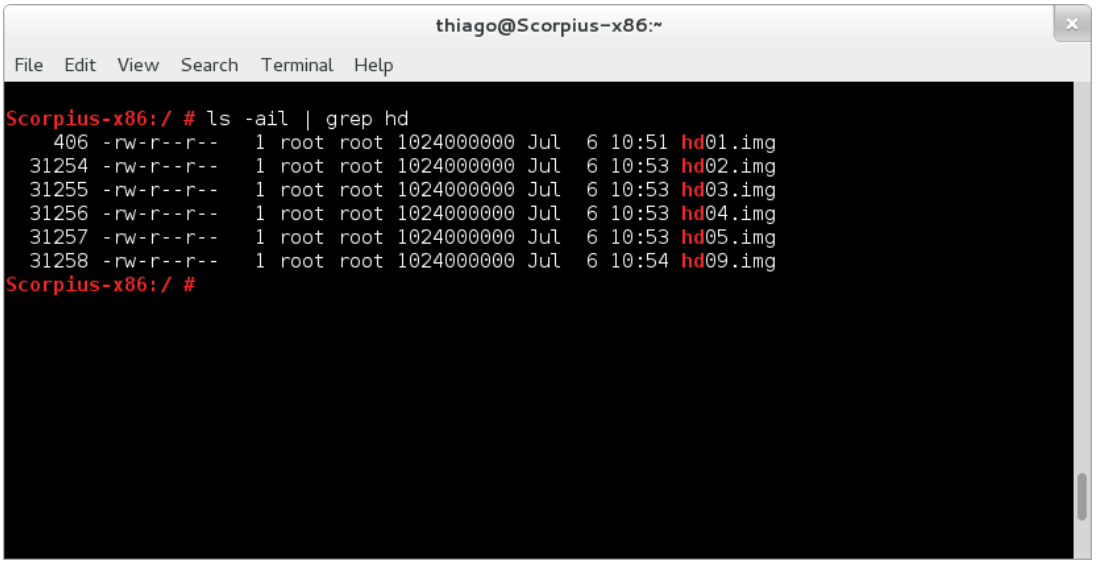

Figura 5.8: Lista das imagens de discos criadas.

Posteriormente, criamos uma partição em cada uma delas usando o comando fdisk que aparece na imagem 5.9.

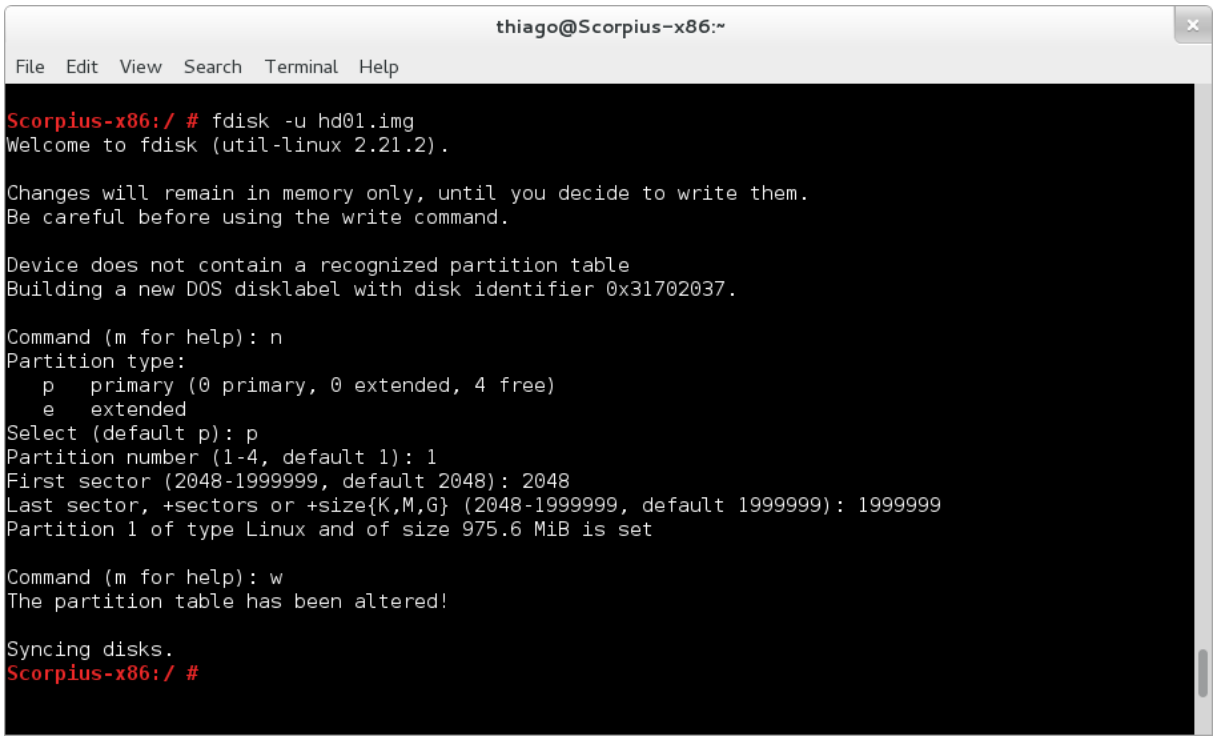

Figura 5.9: Criando partições com o fdisk.

Criado a partição, formatamos cada uma com o comando mkfs.ext3 inserindo um sistema de arquivo EXT3 como está representado na figura 5.10.

Neste ponto, as imagens de disco estavam criadas, nos restou então inserir os arquivos que seriam alvo da futura remoção e recuperação.

Montamos a primeira imagem usando o comando abaixo.

```
# mount hd01.img /mnt/investigation/HD01
```

Preparamos 6 imagens ".jpeg", 6 imagens ".tiff", 6 videos ".mp4", 6 arquivos ".txt"e 6 arquivos ".pdf". Um arquivo de cada tipo por imagem de disco. Os arquivos possuiam marcações especiais que permitiam identificar a qual imagem eles pertenceriam. A figura 5.11 exemplifica as marcações feitas em imagens ".jpeg"e nas ".tiff"referenciando suas respectivas imagens de disco (exceto no caso dos arquivos ".pdf"). As com os número de 1 a 3 são as "jpeg"e as com números 4,5 e 9 são exemplos das ".tiff".

Os arquivos ".txt"simulam uma lista de senhas e o arquivo ".pdf"é um relatório da symantec referente a ameaças de segurança na internet publicado em abril de 2014, também encontrado no site `http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf`.

Segue a tabela com os arquivos inclusos na primeira imagem de disco 5.2.

Por fim, desmontamos a imagem de disco usando o comando a seguir.

```
# umount /mnt/investigation/HD01
```

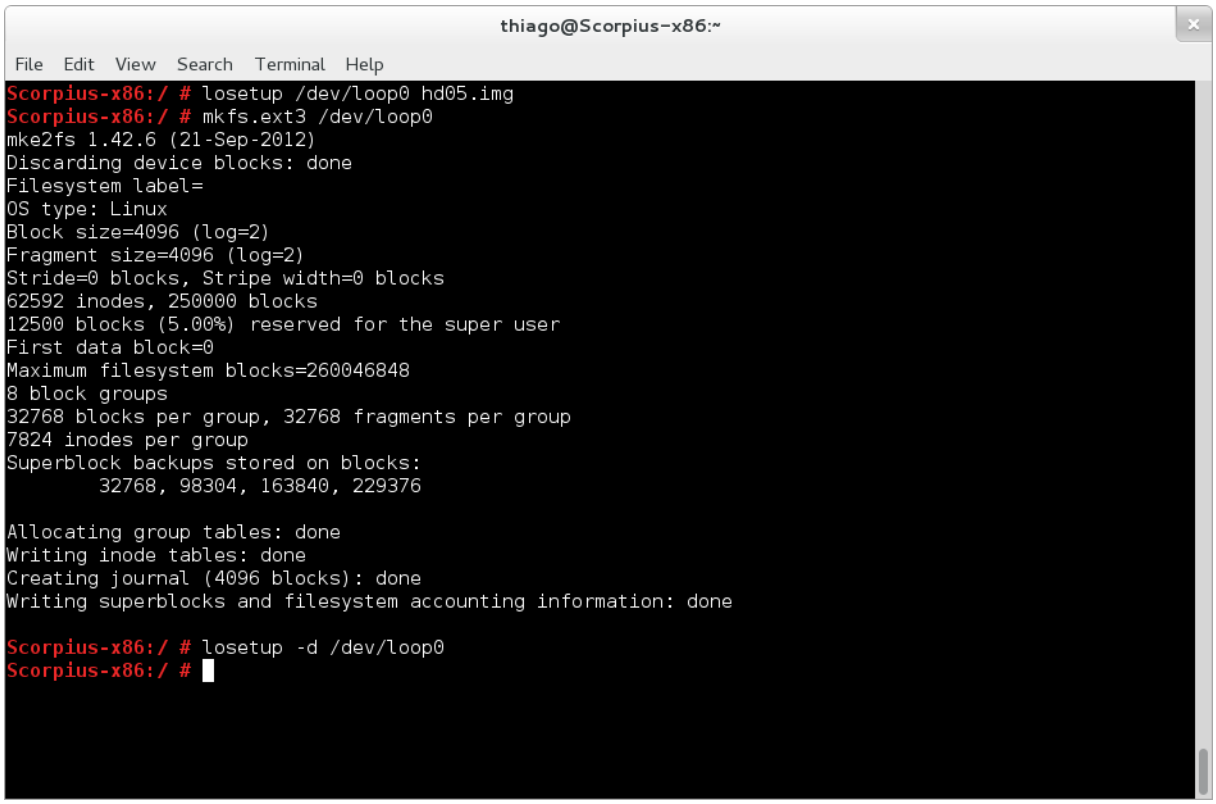

Figura 5.10: Criando um sistema de arquivos ext3 nas imagens.

Nome	Tamanho	I-node	Data
b-istr_main_report_v19_21291018.en-us.pdf	5.2MB	12	Jul 6 15:23
flower_HD01.jpeg	144KB	13	Jul 6 12:23
lenna_HD01.tif	1.1MB	14	Jul 6 12:17
nasa_shuttle_HD01.mp4	1.4MB	15	Jul 6 12:40
pass.txt	8.0K	16	Jul 6 15:20

Tabela 5.2: Arquivos inseridos na primeira imagem de disco.

Figura 5.11: Exemplos das imagens obtidas de `http://www.imageprocessingplace.com/DIP-3E/`, alteradas e colocadas em cada disco. As com numeros de 1 a 3 são ".jpeg"e as demais ".tiff".

Depois repetimos o mesmo processo para cada uma das outras imagens inserindo os seus respectivos arquivos conforme mostram as tabelas 5.3, 5.4, 5.5, 5.6, 5.7.

Nome	Tamanho	I-node	Data
b-istr_main_report_v19_21291018.en-us.pdf	5.2MB	12	Jul 6 15:23
flower_HD02.jpeg	140KB	13	Jul 6 12:23
lenna_HD02.tif	1.1MB	14	Jul 6 12:16
nasa_shuttle_HD02.mp4	2.7MB	15	Jul 6 12:40
pass.txt	8.0KB	16	Jul 6 15:21

Tabela 5.3: Arquivos inseridos na imagem de disco 02.

Nome	Tamanho	I-node	Data
b-istr_main_report_v19_21291018.en-us.pdf	5.2MB	12	Jul 6 15:23
flower_HD03.jpeg	140KB	13	Jul 6 12:23
lenna_HD03.tif	1.1MB	14	Jul 6 12:17
nasa_shuttle_HD03.mp4	5.8MB	15	Jul 6 12:37
pass.txt	8.0KB	16	Jul 6 15:21

Tabela 5.4: Arquivos inseridos na imagem de disco 03.

Nome	Tamanho	I-node	Data
b-istr_main_report_v19_21291018.en-us.pdf	5.2MB	12	Jul 6 15:23
flower_HD04.jpeg	140KB	13	Jul 6 12:24
lenna_HD04.tif	1.1MB	14	Jul 6 12:18
nasa_shuttle_HD04.mp4	9.8MB	15	Jul 6 12:31
pass.txt	8.0KB	16	Jul 6 15:21

Tabela 5.5: Arquivos inseridos na imagem de disco 04.

5.2.2 Procedimentos

Usamos a ferramenta "FRS"em seus 6 níveis para remover os arquivos. O nível 1 foi aplicado nos arquivos que estavam na imagem hd01.img, o segundo nível na segunda imagem e assim por diante.

A tentativa de recuperação dos arquivos foi por meio das ferramentas `"Autopsy"`, `"foremost"` (versão 1.5.7 por J. Kornblum, K. Jendall e N. Mikus), `"Magic Rescue"` (versão 1.1.9 por J. Jensen) e `"ExtUndelete"` (versão 0.2.4 com libext2fs versão 1.42.6 por N E Case), além de alguns conhecimentos de *Data Carving* e as etapas descritas no artigo de [10]. A seguir descreveremos cada etapa realizada.

Nome	Tamanho	I-node	Data
b-istr_main_report_v19_21291018.en-us.pdf	5.2MB	12	Jul 6 15:23
flower_HD05.jpeg	140KB	13	Jul 6 12:24
lenna_HD05.tif	1.1MB	14	Jul 6 12:18
nasa_shuttle_HD05.mp4	12.0MB	15	Jul 6 12:48
pass.txt	8.0KB	16	Jul 6 15:20

Tabela 5.6: Arquivos inseridos na imagem de disco 05.

Nome	Tamanho	I-node	Data
b-istr_main_report_v19_21291018.en-us.pdf	5.2MB	12	Jul 6 15:23
flower_HD09.jpeg	140KB	13	Jul 6 12:26
lenna_HD09.tif	1.1MB	14	Jul 6 12:19
nasa_shuttle_HD09.mp4	47.0MB	15	Jun 16 2007
pass.txt	8.0KB	16	Jul 6 15:20

Tabela 5.7: Arquivos inseridos na imagem de disco 09.

Nível 1

Com a imagem montada no diretório `/mnt/investigation/HD01/`, os comandos de remoção de arquivos usados para esse nível foram:

```
# ./frs -L 1 -c /mnt/investigation/HD01/pass.txt
# ./frs -L 1 -c /mnt/investigation/HD01/nasa_shuttle_HD01.mp4
# ./frs -L 1 -c /mnt/investigation/HD01/lenna_HD01.tif
# ./frs -L 1 -c /mnt/investigation/HD01/flower_HD01.jpeg
# ./frs -L 1 -c /mnt/investigation/HD01/b-istr_main_report_v19_21291018.en-us.pdf
```

Finalizando o processo de remoção dos arquivos da primeira imagem de disco nós desmontamos a unidade com o comando.

```
# umount /mnt/investigation/HD01/
```

A ferramenta FRS, em seu primeiro nível, simplesmente decrementa o número de *links* do *i-node* do arquivo, ou melhor dizendo, o número de entradas de diretórios que apontam para o *i-node* de um arquivo. Quando o valor de *links* chega a 0, então os ponteiros de conteúdo armazenados no *i-node* são limpos, o sistema libera sua região de memória e realiza os procedimentos já explicados anteriormente na seção 3.1.4 desse trabalho. Neste nível da ferramenta não há nenhuma mudança em outros metadados e nem no conteúdo do arquivo.

Nos processos de recuperação de dados assumimos que não sabíamos nada a respeito do disco a ser recuperado e tentamos obter o máximo de informações no sistema analizado.

Para realizar a recuperação em cada nível montamos as imagens com os parâmetros "ro"e "noexec"que indica que elas serão montadas apenas para leitura (*read-only*) e ativando também o parâmetro de não executável (*non-executable*). Portanto o comando de montagem para a primeira imagem de disco ficou assim:

```
# mount -o ro,noexec,loop hd01.img /mnt/investigation/HD01
```

Próximo passo foi utilizar as ferramentas de recuperação de dados mencionadas para tentar recriar os arquivos excluídos.

Neste primeiro passo, usando as ferramentas Magic Rescue e a ferramenta Foremost por meio dos comandos a seguir.

```
# magicrescue -r jpeg-jfif -r jpeg-exif -r avi -d /mnt/Recovery/MRHD01/
/dev/loop0

# foremost -t all -o /mnt/Recovery/TSKHD01/ hd01.img
```

Não obtivemos resultado algum. A ferramenta Magic Rescue nem identificar os arquivos ela identificou. A Foremost reconhece a existência de arquivos, porém todos possuiam tamanhos iguais a zero (não tinham conteúdo) e ainda assim identificou apenas um arquivo no formato ".jpeg"e outro no formato ".pdf".

Porém usando o programa "ExtUndelete", o qual faz uso do arquivo *journal* para realizar a recuperação, facilmente recuperamos todos os arquivos de forma completa. Segue o comando a baixo.

```
# extundelete --restore-all hd01.img
```

A recuperação manual dos arquivos também foi possível com o uso do Autopsy juntamente com o Foremost e o TSK, além de utilitários do *linux*. Baseando nos no artigo escrito por Narvaez [10], realizamos os procedimentos que culminou na recuperação completa dos arquivos.

Primeiramente obtivemos os *i-nodes* dos arquivos excluídos usando a ferramenta `ils`.

```
Scorpiiius-x86:/ # ils -r hd01.img
class|host|device|start_time
ils|Scorpiiius-x86||1404993628
st_ino|st_alloc|st_uid|st_gid|st_mtime|st_atime|st_ctime|st_crtime|st_mode
|st_nlink|st_size
12|f|0|0|1404675519|1404674031|1404675519|0|766|0|0
13|f|0|0|1404675558|1404674031|1404675558|0|766|0|0
14|f|0|0|1404675539|1404674031|1404675539|0|766|0|0
15|f|0|0|1404675548|1404674031|1404675548|0|766|0|0
16|f|0|0|1404675564|1404674031|1404675564|0|766|0|0
```

Listamos o *i-node* do segundo arquivo usando a ferramenta `istat` e descobrimos que tanto ele quanto os outros arquivos excluídos pertencem ao grupo 0 e que os ponteiros para os blocos de cada arquivo foram perdidos com a remoção. Assim usamos o comando `fsstat` para obter as informações sobre a organização do sistema de arquivo e em especial do grupo 0. Segue o comando.

```
Scorpiiius-x86:/ # fsstat hd01.img
[REMOVED]
Group 0:
        Inode Range: 1 - 7824
        Block Range: 0 - 32767
        Layout:
```

```
                Super Block: 0 - 0
                Group Descriptor Table: 1 - 1
                Data Bitmap: 63 - 63
                Inode Bitmap: 64 - 64
                Inode Table: 65 - 553
                Data Blocks: 554 - 32767
        Free Inodes: 7813 (99%)
        Free Blocks: 32208 (98%)
        Total Directories: 2
[REMOVED]
```

Com isso descobrimos que o grupo 0 tem a capacidade de armazenar até 7824 *i-nodes* e que a tabela de *i-nodes* desse grupo possui 489 blocos, o que nos garante 16 *i-nodes* por bloco. Portanto os *i-nodes* de 1 a 16 estarão no primeiro bloco, 17 a 32 no segundo e assim por diante.

Sabendo que o *journal* funciona a nível de blocos, então procuramos neles os dados antigos dos *i-nodes* dos arquivos que queremos recuperar. No nosso caso, o primeiro bloco da tabela de *i-nodes* é o bloco 65. Uma das maneiras de procurar blocos de discos armazenados no arquivo do *journal* é listar as transações pelo seu número sequencial. Para isso podemos usar o programa `jls`, que também faz parte do TSK, limitando a saída usando o comando `grep`. Lembrando que quanto menor o número sequencial da saída, mais antigo é a cópia do bloco de *i-nodes*.

```
Scorpiiius-x86:/ # jls hd01.img | grep -i " 65"
2 :         Unallocated FS Block 65
8 :         Unallocated FS Block 65
15:         Unallocated FS Block 65
24:         Unallocated FS Block 65
33:         Unallocated FS Block 65
41:         Unallocated FS Block 65
49:         Unallocated FS Block 65
```

Assim podemos perceber que a transação 2 do *journal* possui os dados mais antigos do bloco de dados onde os *i-nodes* que estamos procurando residem. A partir dai, precisamos saber o tamanho de cada *i-node* do sistema de arquivos. Essa informação pode ser encontrada no super bloco do sistema de arquivos. Usando o comando `dumpe2fs` pode nos ajudar a encontrar esse dado.

```
Scorpiiius-x86:/ # dumpe2fs hd01.img | grep -i "inode size"
dumpe2fs 1.42.6 (21-Sep-2012)
Inode size:        256
```

Usando `jcat`, o comando `dd` e o comando `xxd` conseguimos visualizar os dados do *i-node* 13. Nos interessa saber os endereços de disco onde o conteúdo do arquivo estão. Essa informação começa no *byte* 40 com os endereços do 12 primeiros blocos do arquivo desejado (até o *byte* 87), e termina nos endereço dos blocos de indireição simples(0x000002e3), dupla (0x00000000) e tripla (0x00000000).

```
Scorpiiius-x86:/ # jcat hd01.img 8 2 | dd bs=256 skip=13 count=1 | xxd
1+0 records in
1+0 records out
```

```
256 bytes (256 B) copied, 0.0406098 s, 6.3 kB/s
0000000: f681 0000 7601 1000 ef9f b953 86a5 b953 ....v......S...S
0000010: ef9f b953 0000 0000 0000 0100 1008 0000 ...S............
0000020: 0000 0000 0100 0000 2004 0000 2104 0000 ............!...
0000030: 2207 0000 2304 0000 2404 0000 2500 0000 "...\#...\$...\%...
0000040: 2604 0000 2704 0000 2804 0000 2904 0000 &...'...(...)...
0000050: 2a04 0000 2b04 0000 e302 0000 0000 0000 *...+...........
0000060: 0000 0000 75d9 73b3 0000 0000 0000 0000 ....u.s.........
0000070: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000080: 1c00 0000 d40b 8955 6ca2 7b65 d090 0165 .......Ul.\{e...e
0000090: ef9f b953 d090 0165 0000 0000 0000 0000 ...S...e........
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000e0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000f0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

Ainda de acordo com o *i-node* recuperado temos um arquivo com 1048950 *bytes* de tamanho(0x00100176). E nos doze primeiros blocos teríamos apenas 49152 *bytes* armazenados ($4096 * 12$). Faltariam ainda 245 blocos que estarão no bloco 0x000002e3. A segui mostramos os 48 endereços iniciais desse bloco.

```
Scorpiiius-x86:/ # dd if=/hd01.img bs=4096 skip=739 count=1 | xxd
0000000: 2c04 0000 2d04 0000 2e04 0000 2f04 0000 ,...-......./...
0000010: 500d 0000 510d 0000 520d 0000 530d 0000 P...Q...R...S...
0000020: 540d 0000 550d 0000 560d 0000 570d 0000 T...U...V...W...
0000030: 580d 0000 590d 0000 5a0d 0000 5b0d 0000 X...Y...Z...[...
0000040: 5c0d 0000 5d0d 0000 5e0d 0000 5f0d 0000 \...]...^..._...
0000050: 600d 0000 610d 0000 620d 0000 630d 0000 `...a...b...c...
0000060: 640d 0000 650d 0000 660d 0000 670d 0000 d...e...f...g...
0000070: 680d 0000 690d 0000 6a0d 0000 6b0d 0000 h...i...j...k...
0000080: 6c0d 0000 6d0d 0000 6e0d 0000 6f0d 0000 l...m...n...o...
0000090: 700d 0000 710d 0000 720d 0000 730d 0000 p...q...r...s...
00000a0: 740d 0000 750d 0000 760d 0000 770d 0000 t...u...v...w...
00000b0: 780d 0000 790d 0000 7a0d 0000 7b0d 0000 x...y...z...{...
[REMOVED]
```

Percebemos assim a existência de alguns conjuntos de blocos consecutivos. Manualmente podemos criar arquivos `".dd"` de cada bloco consecutivo e concatená-los, posteriormente, usando o comando `cat`. Finalmente teríamos um arquivo único com os blocos em sequência mapeados em um único arquivo. Dai poderíamos deixar a cargo do sistema operacional interpretar os dados do arquivo e tentar abri-lo ou usar novamente a ferramenta `foremost` para tentar identificar o tipo de arquivo e realizar a recuperação, como Narvaez fez em seu artigo [10].

Uma das figura recuperadas estão representadas a seguir 5.12.

Os arquivos maiores são mais trabalhosos já que a chance de haver fragmentação é maior. Como é necessário juntar cada conjunto de blocos acabam exigindo mais atenção. Por outro lado, temos acesso aos dados dos blocos onde estão armazenados e a ordem em que eles devem ser unidos. É apenas uma questão de ter tempo bastante para isso. Por fim conseguimos recuperar todos os arquivos sem ter perda de dados.

Figura 5.12: Imagem recuperada usando o procedimento descrito.

Nível 2

Montamos a segunda imagem no diretório `/mnt/investigation/HD02/`. Usamos os seguintes comandos de remoção de arquivos.

```
# ./frs -L 2 -vc /mnt/investigation/HD02/pass.txt
# ./frs -L 2 -vc /mnt/investigation/HD02/nasa_shuttle_HD02.mp4
# ./frs -L 2 -vc /mnt/investigation/HD02/lenna_HD02.tif
# ./frs -L 2 -vc /mnt/investigation/HD02/flower_HD02.jpeg
# ./frs -L 2 -vc /mnt/investigation/HD02/b-istr_main_report_v19_21291018.en-us.pdf
```

Para finalizar o processo de remoção dos arquivos dessa imagem de disco nós desmontamos a unidade com o comando.

```
# umount /mnt/investigation/HD02/
```

Em seu segundo nível a ferramenta simplesmente altera o nome do arquivo por valores aleatórios. Como o sistema de arquivo já insere zeros nos campos de endereços dos blocos dos i-nodes e modifica outras informações, como o tamanho do arquivo, basicamente tentamos evitar a busca por um arquivo específico meio do seu nome. Porém quando existe uma ferramenta capaz de analisar o arquivo *journal*, esperamos que a informação destruída possa ser recuperada simplesmente varrendo o *journal* em busca do bloco correspondente ao diretório alterado (lembramos aqui que nos *Unix* e nos *Unix-like* os nomes de arquivos são armazenados nas entradas de diretórios). Afinal de contas, quando alteramos os dados de um arquivo, o bloco alterado pode ser armazenado no *journal*.

Rodamos então novamente a ferramenta *ExtUndelete* usando o mesmo comando do caso anterior.

```
# extundelete --restore-all hd02.img
```

E , de fato, obtivemos como resultado a recuperação dos arquivos como um todo, exceto alguns dos nome destes. O primeiro arquivo recuperado possuía o nome correto, mas os demais apareciam com nomes diferentes. Porém o conteúdo de cada arquivo estavam perfeitos. Usando o Autopsy, há perdas de nomes porém não de todos os arquivos. Dependendo da ordem de exclusão dos arquivos o número pode variar como veremos na imagem 5.13. Dos 5 arquivos removidos, apenas 1 nome permaneceu sem alteração. Isso é devido ao comportamento a função `rename()`, como explicaremos a seguir.

Portanto mesmo alterando os nomes dos arquivos (de certa maneira estaríamos quebrando uma ligação entre a estrutura de metadado e a entradas de diretório do arquivo) é possível recuperar os arquivos após a sua remoção. Àqueles que desejam nível de segurança maior, esse ainda está longe de ser o indicado.

Nível 3

Iniciamos o próximo nível montando a terceira imagem de disco no diretório `/mnt/investigation/HD03/`. Usamos os seguintes comandos de remoção de arquivos.

```
# ./frs -L 3 -v -j /dev/loop0 /mnt/investigation/HD03/pass.txt
# ./frs -L 3 -v -j /dev/loop0 /mnt/investigation/HD03/nasa_shuttle_HD03.mp4
# ./frs -L 3 -v -j /dev/loop0 /mnt/investigation/HD03/lenna_HD03.tif
# ./frs -L 3 -v -j /dev/loop0 /mnt/investigation/HD03/flower_HD03.jpeg
# ./frs -L 3 -v -j /dev/loop0 /mnt/investigation/HD03/b-istr_main_report_v19_21
 291018.en-us.pdf
```

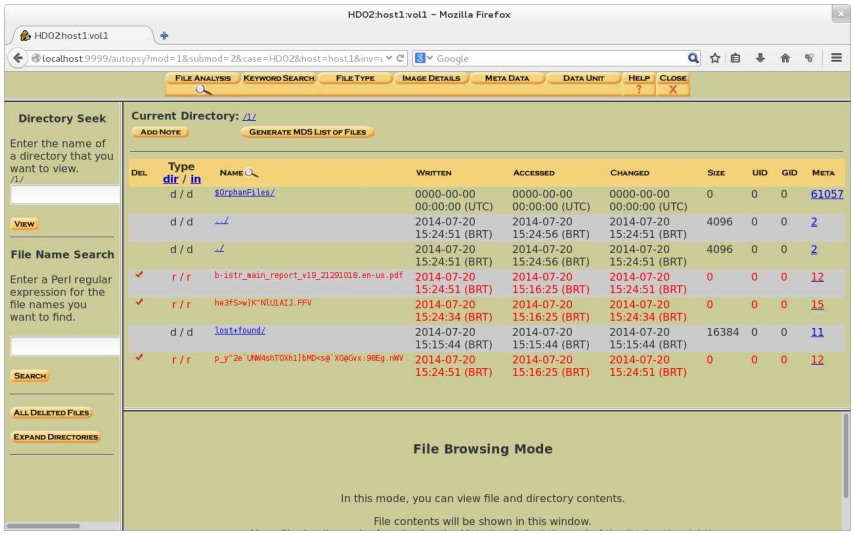

Figura 5.13: Nomes destruídos pela ferramenta FRS, mas ainda assim não houve dificuldade em realizar a recuperação dos arquivos.

Finalizamos o processo de remoção dos arquivos dessa imagem de disco desmontando a unidade com o comando.

```
# umount /mnt/investigation/HD03/
```

No terceiro nível da ferramenta FRS, realizamos o mesmo que foi feito nos níveis anteriores e acrescentamos mais dois passos interessantes. Após alterar o nome do arquivo, copiamos dados verdadeiros (existente no próprio sistema de arquivos) de outro *i-node* para os campos daquele que estamos destruindo. Posteriormente sobrescrevemos no máximo os 32768 blocos de dados do arquivo `journal` com zeros. Além disso, alteramos os dados do números de blocos, valor do bloco de início, número sequencial da primeira transação e bloco da primeira transação do *journal* - todos estes dados estão no superbloco do *journal*. Por fim realizamos a exclusão do arquivo. Com isso esperávamos que o sistema não conseguisse recuperar os nomes dos arquivo nem os seus dados.

Os resultados de certa forma nos surpreendeu. Os arquivos realmente não foram recuperados quando usamos as ferramentas de recuperação em seus modos automáticos. Porém elas ainda nos mostravam os nomes de alguns arquivos removidos pela FRS, algo totalmente inesperado.

Observando o comportamento do sistema operacional ao realizar a chamada da função `rename()`, modo como realizamos a alteração do nome. Descobrimos que ele não sobrescreve no disco a entrada que solicitamos modificar. Ao invés disso, aloca uma nova entrada de diretório e marca a anterior como disponível, mantendo os dados da antiga. Quando removemos o arquivo, o sistema também marca a nova entrada criada como disponível.

O processo de sobrescrita ocorre quando realizamos o procedimento com os outros arquivos pertencentes ao mesmo diretório. Eventualmente as antigas entradas já desalocadas são sobrescritas quando roda esse procedimento de nova alocação. Porém caso não seja feita nenhuma outra operação de arquivos, o nome do arquivo que removermos aparecerá na entrada de diretório em que ele residia. A solução para esse problema está

em realizar a alteração direto no arquivo de bloco que mapeia o disco, do mesmo modo que procedemos quando destruímos os *i-nodes* e o *journal*.

Além disso, usando o Autopsy é possível varrer o disco que estamos analisando e permite que vejamos o que está gravado no disco interpretando seus dados, como texto ASCII. Juntando informações de tamanho de bloco, as assinaturas de arquivos e o modo como os dados estão espalhados em disco (comando `dumpe2fs` por exemplo) permite que separemos as partes de arquivos. Contando ainda com o fato de que o sistema Unix tende a manter os dados de um arquivo o mais próximo um do outro para evitar os seeks nos discos, é possível remontar os arquivos legíveis, como por exemplo os ".pdf", ou os ".txt". A recuperação de video e imagens já são mais complexas e dependendo da fragmentação delas, o uso desse nível da ferramenta FRS já torna bem difícil a recuperação de dados. A figura 5.14 mostra o conteúdo de um arquivo que poderia ser recuperado. E a figura 5.15 mostra o início de um arquivo JPEG identificado pela sua assinatura 0XFFD8 e o identificador 0x4A46494600 ou seja JFIF (*JPEG File Interchange Format*) escrito do 7º ao 10º *byte*.

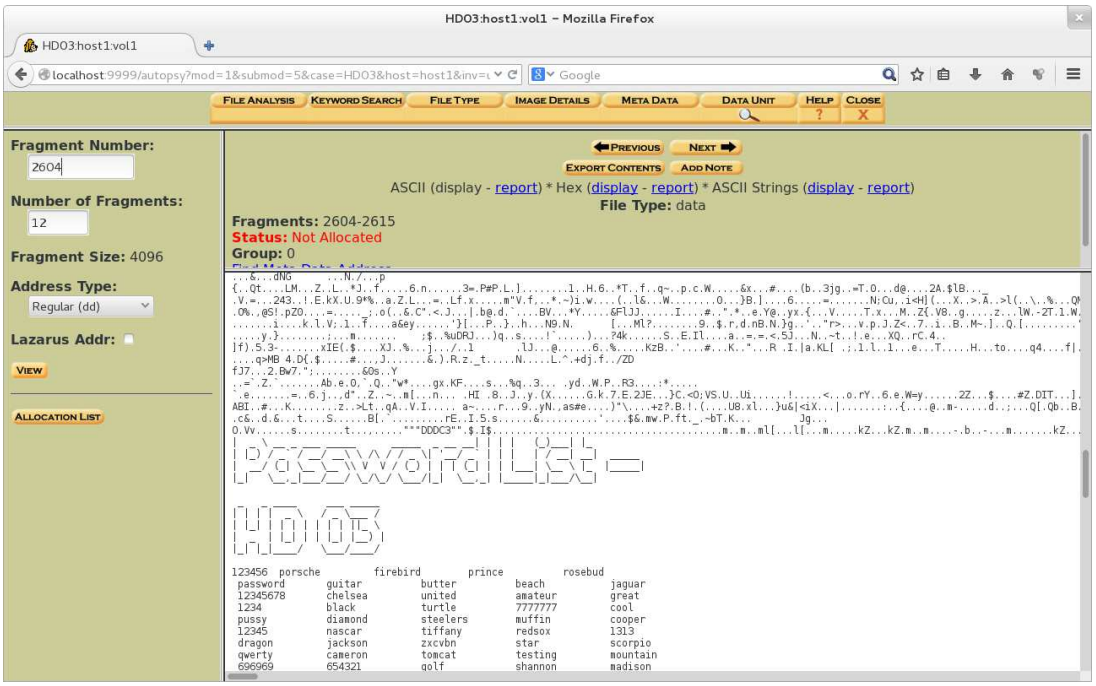

Figura 5.14: Conteúdo de 12 blocos do disco 03 em análize mostrando dados de um arquivo que possui conteúdo legível em seus dados em disco.

Nível 4

No quarto nível montamos a imagem de disco correpondente no diretório `/mnt/investigation/HD04/`. Usamos os seguintes comandos de remoção de arquivos.

```
# ./frs -L 4 -v -j /dev/loop0 /mnt/investigation/HD04/pass.txt
# ./frs -L 4 -v -j /dev/loop0 /mnt/investigation/HD04/nasa_shuttle_HD04.mp4
# ./frs -L 4 -v -j /dev/loop0 /mnt/investigation/HD04/lenna_HD04.tif
```

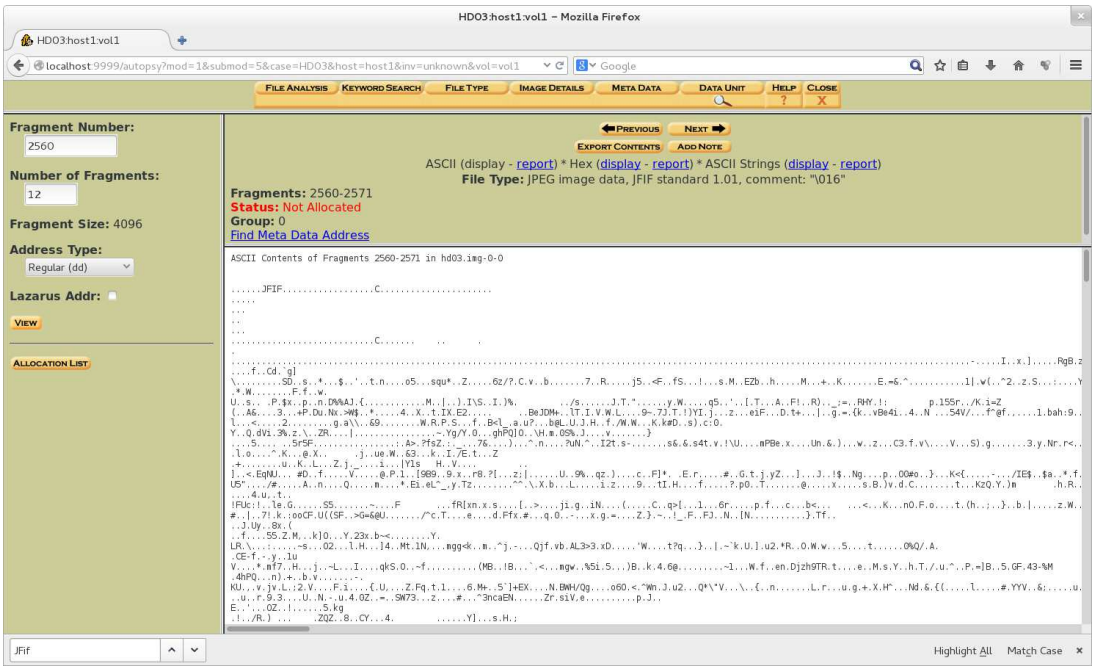

Figura 5.15: Conteúdo de 12 blocos do disco 03 em análize mostrando o início de um arquivo jpeg identificado pela assinatura 0xFFD8 nos dois primeiros *bytes*.

```
# ./frs -L 4 -v -j /dev/loop0 /mnt/investigation/HD04/flower_HD04.jpeg
# ./frs -L 4 -v -j /dev/loop0 /mnt/investigation/HD04/b-istr_main_report_v19_212
 91018.en-us.pdf
```

Após esses procedimentos, desmontamos a imagem de disco com o comando.

```
# umount /mnt/investigation/HD04/
```

Neste nível a FRS realiza os mesmo passo do nível 3, onde evitamos a tentativa de recuperação dos dados usando o *i-node* e os blocos do *journal*, além de destruir o primeiro bloco de conteúdo do arquivo, inserindo zeros em todo ele. Esse último passo tem por objetivo destruir as assinaturas que identificam tipos de arquivos, elementos usados em processos de data *carving*, bem como dados importantes a respeito da sua estrutura. Por exemplo, se fosse uma imagem, não saberíamos qual a sua resolução ou a versão do tipo de arquivo. Esperávamos dessa maneira que limitássemos mais ainda a capacidade de recuperação das ferramentas.

E de fato, os arquivos que dependem do primeiro bloco para serem recuperados e abertos (investigações que usam *data carving*), não foram. Verificamos que arquivos que possuem textos no seu conteúdo , ou melhor arquivos cujo conteúdo são legíveis, como ".html", ".pdf", programas fontes, entre outros, que possuíam mais de 4kb podem ainda ter parte de suas informações visualizadas. Nesse estágio não foi possível remontar ou recuperar arquivo algum. No geral isso se deu pelos campos sobrescritos e em grande parte pela fragmentação dos arquivos.

Por outro lado, o arquivo de senhas ".txt", foi possível ver claramente o conteúdo não sobrescrito, semelhante ao que ocorreu no nível anterior. Arquivos menores que 4kB foram

destruídos e a suas informações não puderam ser recuperadas. Nenhum dos arquivos de imagens e de vídeos foi possível recuperar. O que vemos em disco aparenta mais ser dados sem valor algum a dados de um dos arquivo que lá estavam anteriormente. Sabemos que existem partes de arquivos na imagem montada, porém devido a fragmentação dos dados e a destruição da estrutura que permitiria interpretá-los, sua remontagem não foi possível.

O fato de colocar zeros em todo um bloco inviabiliza a obtenção de informações importantes, porém deixa marcas claras de onde possivelmente existiria o início de um arquivo. Em um caso mais crítico, poderíamos supor, por exemplo, que o usuário esteja tentando ter acesso a uma imagem e que existam ferramentas capazes de mostrar pedaços de uma suposta imagem (mesmo sem conhecer seu cabeçalho e preenchendo os campos destruídos com branco ou preto). Poderíamos supor isso para qualquer tipo de arquivo porém a situação com imagens fica mais fácil de ser compreendida. Neste caso, a existência de exatamente um bloco inteiro com zeros faz dele um excelente alvo para marcar o início de um arquivo. Apesar de toda dificuldade em identificar esse início, teriamos como consequência direta a diminuição no campo de busca por parte do atacante.

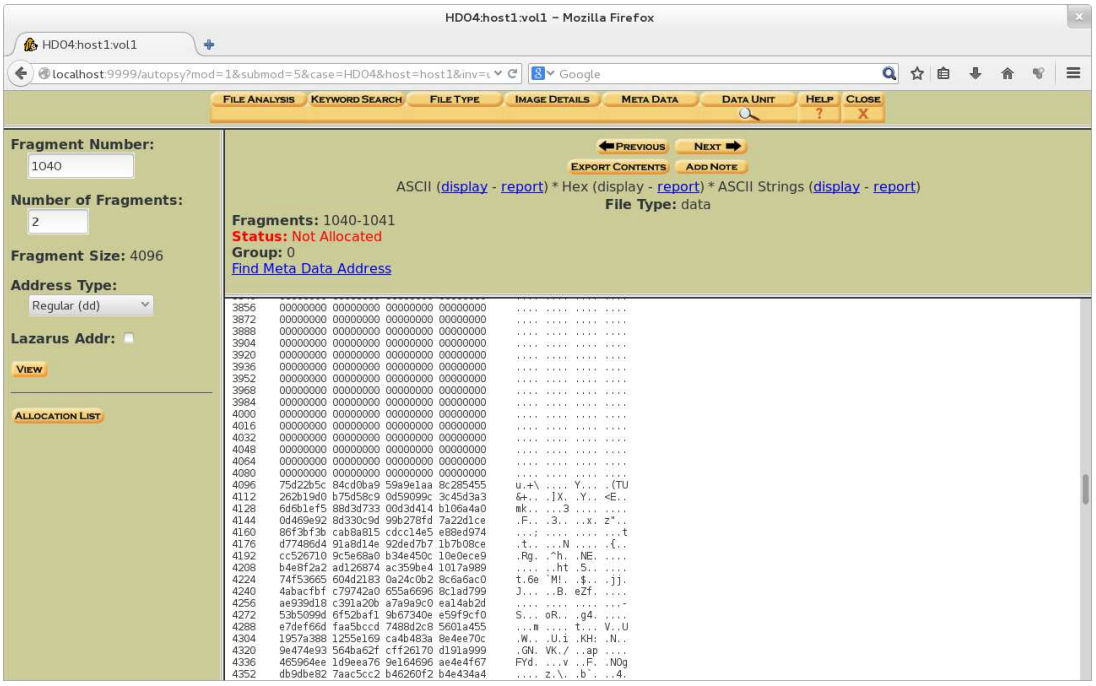

Figura 5.16: Conteúdo do arquivo ".jpeg"depois da sua exclusão.

Assim a ferramenta apresenta o seu próximo nível. Capaz ao invés de escrever zeros no primeiro bloco, escrever valores aleatórios usando a entropia do sistema como explicamos anteriormente.

Nível 5

No quinto nível montamos sua imagem de disco no diretório `/mnt/investigation/HD05/`. Usamos os seguintes comandos de remoção de arquivos.

```
# ./frs -L 5 -v -j /dev/loop0 /mnt/investigation/HD05/pass.txt
# ./frs -L 5 -v -j /dev/loop0 /mnt/investigation/HD05/nasa_shuttle_HD05.mp4
```

```
# ./frs -L 5 -v -j /dev/loop0 /mnt/investigation/HD05/lenna_HD05.tif
# ./frs -L 5 -v -j /dev/loop0 /mnt/investigation/HD05/flower_HD05.jpeg
# ./frs -L 5 -v -j /dev/loop0 /mnt/investigation/HD05/b-istr_main_report_v19_212
 91018.en-us.pdf
```

Após esses procedimentos, desmontamos a imagem de disco com o comando.

```
# umount /mnt/investigation/HD05/
```

A única mudança em relação ao nível anterior é a sobrescrita em mais 2 blocos, além do primeiro, e o fato dos dados usados para tal serem aleatórios (no nível 4 usava-se apenas zeros). Desde o último nível já não conseguimos realizar a recuperação de nenhum arquivo de forma completa. O tempo médio de execução da ferramenta para para um arquivo de 50MB foi cerca de 2,4s. Abaixo na figura 5.17 vemos o conteúdo dos dados armazenados no terceiro e quarto blocos de um arquivo "jpeg"excluído neste nível.

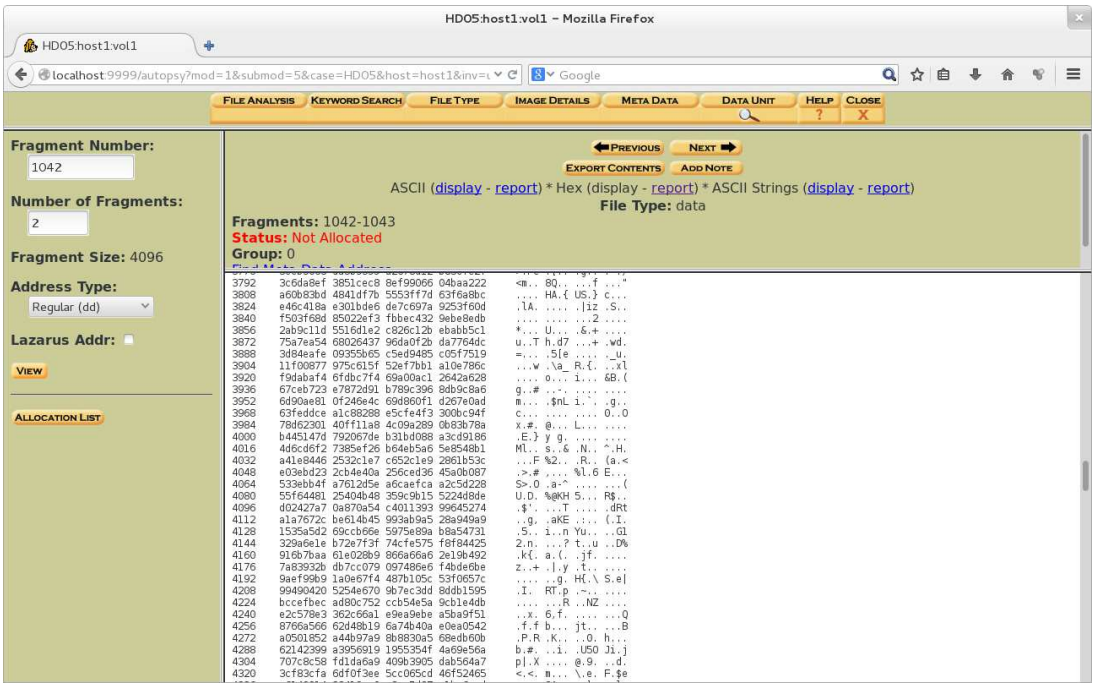

Figura 5.17: Conteúdo do arquivo ".jpeg"depois da sua exclusão.

Para melhor percepção, abaixo mostramos dois *reports*, um antes e outro depois da execução da ferramenta FRS. Observando os *bytes* e sua representação ASCII, principalmente na junção de um bloco com o seguinte, percebemos que quase não há diferenças de padrões no conteúdo dos blocos do arquivo apontado pelo *i-node* 13 no primeiro *report*, para o que apresentamos logo em seguida. Teoricamente, como a aleatoriedade permanece no blocos anteriores (primeiro e segundo bloco), a dificuldade de se recuperar um arquivo nestas condições aumentam drasticamente, na medida em que não se consegue identificar o seu início.

```
                    Autopsy hex Fragment Report
------------------------------------------------------------------------
```

```
                    GENERAL INFORMATION
Fragments: 1042-1043
Fragment Size: 4096

Pointed to by Inode: 13
Pointed to by files:
  /1/flower_HD05.jpeg
MD5 of raw Fragment: 0637411206d38ec95edfc8c99eadc257  -
MD5 of hex output: 3c4b93c095d330c44d9106abb22fa984  -

Image: '/var/lib/morgue/HD05/host1/images/hd05.dd'
Offset: Full image
File System Type: ext

Date Generated: Sun Jul 20 16:59:59 2014
Investigator: unknown
------------------------------------------------------------------
                        CONTENT
[REMOVED]
4000        11d02192 3dba2504 e96131cd d9bd3f7d         ..!. =.%. .a1. ..?}
4016        b75a97cf da0f7433 7d4a7e84 47614f9f         .Z.. ..t3 }J~. GaO.
4032        cd7cc210 ae00d319 5222e9fa ef6e77f5         .|.. .... R".. .nw.
4048        cc36f549 71f5dd39 e5c85754 b525aead         .6.I q..9 ..WT .%..
4064        00bc714f 0e238b18 e3efd65e bea75116         ..qO .#.. ...^ ..Q.
4080        4e676682 ddad3432 68334b29 3c95aa03         Ngf. ..42 h3K) <...
4096        d02427a7 0a870a54 c4011393 99645274         .$'. ...T .... .dRt
4112        a1a7672c be614b45 993ab9a5 28a949a9         ..g, .aKE .:.. (.I.
4128        1535a5d2 69ccb66e 5975e89a b8a54731         .5.. i..n Yu.. ..G1
4144        329a6e1e b72e7f3f 74cfe575 f8f84425         2.n. ...? t..u ..D%
4160        916b7baa 61e028b9 866a66a6 2e19b492         .k{. a.(. .jf. ....
[REMOVED]
------------------------------------------------------------------
                    VERSION INFORMATION
Autopsy Version: 2.24
The Sleuth Kit Version: 4.1.0
```

Segue o segundo *report*.

```
                  Autopsy hex Fragment Report
------------------------------------------------------------------
                    GENERAL INFORMATION
Fragments: 1042-1043
Fragment Size: 4096
Not allocated to any meta data structures
MD5 of raw Fragment: fa13213af82941340415704ebd635fa6  -
MD5 of hex output: e1e688d16ce627997496cc85978a6d55  -

Image: '/var/lib/morgue/HD05/host1/images/hd05.dd'
Offset: Full image
File System Type: ext

Date Generated: Sun Jul 20 17:00:55 2014
Investigator: unknown
------------------------------------------------------------------
                        CONTENT
[REMOVED]
```

```
4000        1261c24c cc68b8b4 10bd8225 9bb2aef3        .a.L .h.. ...% ....
4016        8b4e6ad1 50364216 97e703aa 87f4dc56        .Nj. P6B. .... ...V
4032        5d1c5db9 d06ad62b 5ba8a972 6dc12058        ].]. .j.+ [..r m. X
4048        a4c6ddd9 ab9ed48b df3645b0 ee139c41        .... .... .6E. ...A
4064        c8887297 34edd7c3 a82e2ded 6df33691        ..r. 4... ..-. m.6.
4080        1c22b353 98ef557c 646102d2 40794d40        .".S ..U| da.. @yM@
4096        d02427a7 0a870a54 c4011393 99645274        .$'. ...T .... .dRt
4112        a1a7672c be614b45 993ab9a5 28a949a9        ..g, .aKE .:.. (.I.
4128        1535a5d2 69ccb66e 5975e89a b8a54731        .5.. i..n Yu.. ..G1
4144        329a6e1e b72e7f3f 74cfe575 f8f84425        2.n. ...? t..u ..D%
4160        916b7baa 61e028b9 866a66a6 2e19b492        .k{. a.(. .jf. ....
[REMOVED]
--------------------------------------------------------------------------
                    VERSION INFORMATION
Autopsy Version: 2.24
The Sleuth Kit Version: 4.1.0
```

Assim como no nível anterior, a não ser com relação ao nome de um ou dois arquivos, com a tecnologia disponível para recuperação ao nosso alcance, não foi possível realizar recuperação de arquivo algum. Devemos frisar que arquivos com tamanhos maiores que 12kb no formato ".txt"ou outros que possuem seus conteúdos no formato ASCII, podem ter seu conteúdo violado sem maiores dificuldades. Basta realizar busca por conteúdo ASCII usando palavras chaves em uma ferramenta como o AUTOPSY junto com o TSK por exemplo.

Nível 9

No último nível que a ferramenta opera, montamos sua imagem de disco no diretório `/mnt/investigation/HD09/`. Usamos os seguintes comandos de remoção de arquivos.

```
# ./frs -L 9 -v -j /dev/loop0 /mnt/investigation/HD09/pass.txt
# ./frs -L 9 -v -j /dev/loop0 /mnt/investigation/HD09/nasa_shuttle_HD09.mp4
# ./frs -L 9 -v -j /dev/loop0 /mnt/investigation/HD09/lenna_HD09.tif
# ./frs -L 9 -v -j /dev/loop0 /mnt/investigation/HD09/flower_HD09.jpeg
# ./frs -L 9 -v -j /dev/loop0 /mnt/investigation/HD09/b-istr_main_report_v19_212
 91018.en-us.pdf
```

Após esses procedimentos, desmontamos a imagem de disco com o comando.

```
# umount /mnt/investigation/HD09/
```

Cada bloco de conteúdo do arquivo é sobrescrito com valores aleatórios. Os metadados passam pelos processos de destruição já mencionado nos níveis anteriores e o arquivo *journal* também perde dados de até 32768 blocos. Nessa configuração não foi possível recuperar informação alguma que fosse relevante. A única opção restante para realizar a recuperação seria avaliar fisicamente os 0's e 1's do disco com o auxilio de um microscópio eletrônico e apostando em métodos estatísticos.

Dr. Cohen, (Fred Cohen), presidente do Instituto de Ciências da Califórnia, onde ele lidera um programa de forense digital. É CEO (*Chief Executive Officer*) da Fred Cohen & Associates, uma empresa que faz trabalhos de perícia que inclui análise forense digital, presta consultoria em proteção de informações, escreveu um artigo [2] a respeito

da recuperação de arquivos. Ele iniciou um projeto em 2007, para realmente testar se os dados podem ser recuperados mesmo após ser sobrescrito uma vez a partir de um disco rígido com o uso de um microscópio eletrônico. Nesse artigo ele conclui que já é consensual, dentro de algumas comunidades de especialistas na *web*, não ser possível recuperar dados sobrescritos em discos rígidos modernos.

O assunto é polêmico pois, de acordo com [21], existe a crença de que quando um bit 1 é gravado em um disco, onde anteriormente existia o valor 0, teríamos como efeito real um valor lido próximo de 0,95 e quando um bit 1 é gravado sobre outro bit 1, esse valor é próximo de 1,05. Porém, em seu artigo Cohen conclui que isso não é válido para os disco mais atuais e que as chances de se conseguir acertar o dado que estava realmente guardado em um campo antes de ocorrer a alteração é praticamente o mesmo que jogar uma moeda e descidir 0 para cara e 1 para coroa. Portanto apenas sobrescrever os dados uma única vez seria o suficiente para tornar dados inacessíveis.

Como a sobrescrita de dados já resolve o nosso problema de acesso indevidos a dados, porque então procurar outras maneiras de remover dados de forma segura? A resposta está no custo que essa operação tem para ser realizada. Assim na próxima seção veremos como a FRS se comportou em relação ao seu tempo de exclusão.

Tempo de Remoção

Nesta seção comparamos o tempo de remoção da ferramenta FRS com três software bastante usados para remover arquivos. Um deles é o "remove - rm"que apenas realiza a exclusão de arquivos, o segundo é o "secure remove - srm"e o outro é o "dd", muito usado para sobrescrita de dados. A tabela 5.8 resume os resultados, que também são mostrados nos gráficos abaixo cujo o tempo foi medido usando o comando `time` do linux.

Ferramentas	**Tamanho dos Arquivos Removidos**		
	10MB	**100MB**	**1GB**
rm	$0,023s$	$0,025s$	$0,021s$
FRS-L1	$0,002s$	$0,002s$	$0,003s$
FRS-L2	$0,116s$	$0,114s$	$0,132s$
FRS-L3	$2,379s$	$2,483s$	$2,839s$
FRS-L4	$2,410s$	$2,526s$	$2,602s$
FRS-L5	$2,415s$	$2,407s$	$2,515s$
FRS-L9	$2,722s$	$3,813s$	$22,322s$
dd	$0,017s$	$0,302s$	$14,093s$
srm -s	$164,826s$	+ de $600,000s$	+ de $600,000s$

Tabela 5.8: Tempos de remoção de arquivos na execução da ferramenta FRS juntamente com o do software "dd"e "rm".

Com relação aos tempos de remoção onde não há sobrescrita dos dados, verificamos que a FRS em seu nível 1 de execução é bem mais rápido que o rm. Isso porque ela é uma ferramenta limitada e acaba não realizando muitas verificações operações para remover um arquivo. Percebemos também que quando utilizamos o seu nível 2 o seu tempo se torna quase 5 vezes superior ao do rm. De toda maneira, para todos os tamanhos, testados, em ambos os casos levou-se menos de 0,5s para finalizar a remoção 5.18.

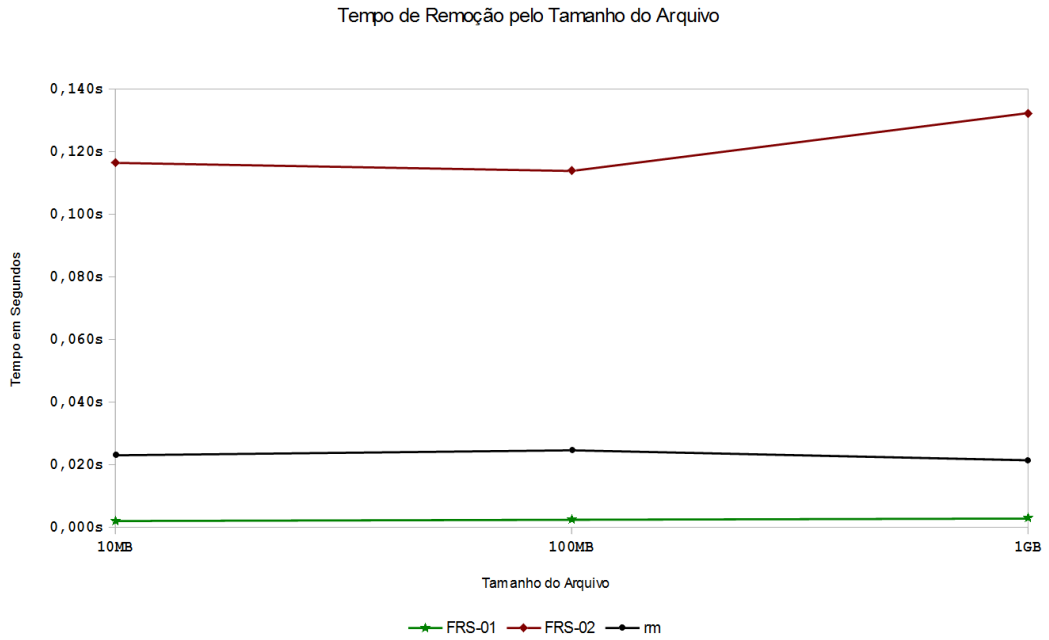

Figura 5.18: Tempo gasto em segundos para a remoção de arquivos usando a FRS .

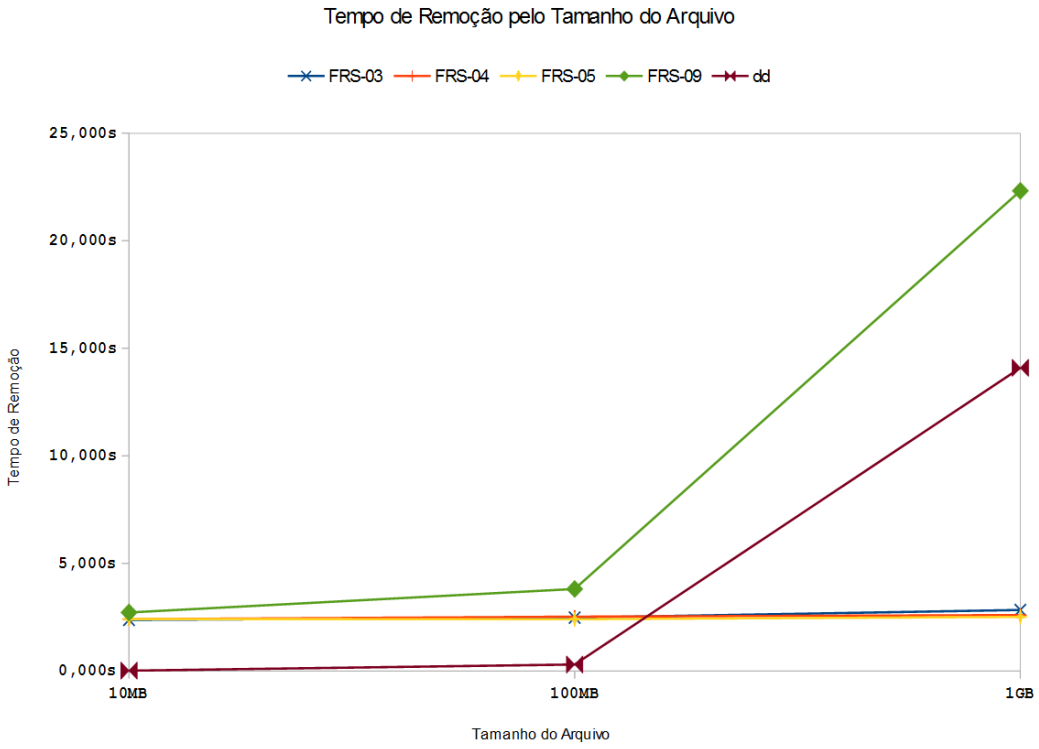

Figura 5.19: Tempo gasto em segundos para a remoção de arquivos usando a FRS .

Nos níveis 3, 4 e 5, a FRS já realiza sobrescrita no *journal*, no *i-node* e nos primeiros blocos de conteúdo do arquivo. Desses passos, o que mais demanda processamento é sem dúvida a sobrescrita do journal. Tanto é que os três possuem tempos muito parecidos como vemos no gráfico 5.19.

E quando a ferramenta utiliza sua capacidade de destruição mais alta, percebemos um crescimento enorme no tempo de execução. Para o arquivo de 1GB, o tempo de remoção gasto no estágio 9 é 887,6% maior quando comparado com o estágio imediatamente anterior (5 da ferramenta). A sobrescrita pela FRS leva de 4s a 8s de tempo a mais que o software `dd`. Se a compararmos com a "srm", o tempo de execução da FRS nível 9 não chega a 5% do tempo gasto pela "srm"quando executada no modo "simple pass (parametro -s)". Por padrão, a "srm"usa o método de Gutmann para realizar as sobrescritas de conteúdo. Além disso a ferramenta renomeia o arquivo (da mesma maneira que FRS faz) e também atribui zero ao tamanho do arquivo. No modo *simple pass*, com o parâmentro -s, essa ferramente executa apenas um passo de sobrescrita usando zeros. Dessa maneira o tempo médio da FRS ainda pode ser considerado aceitável, já que além de cuidar do conteúdo do arquivo, a ferramenta lida com o "journal"e as outras estruturas do sistema. Ideal mesmo seria se o programa realizasse as sobrescritas dos conteúdos em "background"e preferencialmente quando o sistema estivesse ocioso. Algo que ainda não faz.

Considerando as operações que realiza, acreditamos que a ferramenta possui um tempo aceitável principalmente quando olhamos para o tempo dos níveis 4 e 5 e consideramos as possibilidades de recuperação de arquivos já mencionadas.

Capítulo 6

Conclusões

Este trabalho mostrou que existem diversas maneiras de se remover um arquivo. De modo geral a remoção de arquivos não passa de uma abstração em camadas mais altas a respeito do que o usuário deseja ver. No fundo os dados continuam armazenados e acessíveis nos dispositivos físicos.

Como é observável pelos experimentos realizados, as ferramentas "Magic Rescue"e a ferramenta "Foremost"usadas em modo automático não se demonstraram eficientes na recuperação de arquivos em sistemas EXT3. Apesar de identificar a existência de arquivos removidos, elas não foram capazes de realizar a recuperação como fez a ferramenta "ExtUndelete".

Percebemos que mesmo alterando os nomes dos arquivos (ou quebrando a ligação existente entre as entradas de diretórios e os *inodes*) é possível recuperá-los após a sua remoção. Isso porque é possível encontrar os *i-nodes* mesmo sem a entrada de diretório e também pelo fato dados poderem ser recuperados usando o arquivo *journal*.

A limpeza de dados do arquivo *journal* se mostrou bastante agressiva em termos de ataque aos procedimentos automatizados de recuperação já existentes. Porém não impede o acesso direto ao conteúdo dos arquivo que ainda permanecem nas mídias. Com paciência e tendo tempo para realizar a recuperação, essa tarefa pode ser feita como se fosse um quebra cabeça, porém um tanto mais complexo.

A ferramenta FRS apresentou vários problemas e limitações, mas mostrou que é possível dificultar a recuperação de dados de forma simples e rápida realizando operações de sobrescrita a nível de estrutura do sistema de arquivo EXT3. Seu desempenho atendeu as nossas expectativas com relação ao tempo de execução principalmente se levarmos em conta as limitações tecnológicas de escrita em disco. Após seu nível 4 evitou a recuperação de arquivos e manteve o sistema consistente.

Por fim, concluímos que para se inviabilizar a recuperação de arquivos, basta sobrescrever seu conteúdo. Se a performance do sistema para isso for muito prejudicada, podemos dificultar a recuperação limpando alguns blocos do arquivo *journal* e os primeiros blocos de conteúdo dos arquivos. Isso já é suficiente para limitar a atuação de boa parte dos programas de recuperação de dados e não ter muito custo para o sistema de arquivos.

6.1 Trabalhos Futuros

Após a conclusão deste trabalho, algumas sugestões e propostas para trabalhos futuros foram levantadas e ficaram registradas neste espaço para futuras consultas.

Arquivos pequenos - menores que 50kb quando não sobrescritos são muito vulneráveis a procedimentos de *Data Carving*, logo se realmente forem arquivos que não podem ser visto por terceiros, é essencial a sua sobrescrita. É inevitável o desenvolvimento de algoritmos já embutidos no sistema operacional com esse intuito para manter elevado o nível de segurança dos dados.

Após operações em arquivos sensibilizarem o disco, sobrescrever o *journal* pode ser uma saída para dificultar o acesso a dados sensíveis. A sobrescrita de parte do conteúdo do *journal* durante o desligamento do sistema além de reduzir o espaço ocupado por esse arquivo, poderia dar mais segurança ao sistema.

Com relação a ferramenta FRS, identificamos muitas interferências do sistema operacional nas suas operações. Concluímos que a maneira mais eficiente de executá-las é processando diretamente o arquivo que mapeia o dispositivo de bloco. Ficou também evidente que o maior custo em todas as operações está relacionado a sobrescrita de blocos no disco. Logo, a execução da ferramenta em segundo plano, de preferência quando o sistema está ocioso, deixaria menos perceptível a demora nos procedimentos.

Por fim, concluímos que é possível aumentar a segurança dos dados que passaram por processos de remoção nos computadores. Reforçamos que informações sensíveis ainda precisam de muitos cuidados ao serem descartadas. Porém existem preocupações a esse respeito pelos fabricantes de *hardware*. Como no caso dos discos SSD que aceitam comandos de limpeza (como o TRIM) dos sistemas operacionais. Ideal seria que os desenvolvedores compartilhassem dessa ideia e agregem novas funcionalidades aos sistemas.

Appendices

Apêndice A

O Código da FRS

Arquivo: ./src/frs.c

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <stdlib.h>
5  #include <signal.h>
6  #include <string.h>
7
8  #include "frs.h"
9  #include "file.h"
10
11 #define _FILE_OFFSET_BITS 64
12 #define _LARGEFILE64_SOURCE
13
14 /* global vars */
15 struct opt_t options;
16 struct sigaction sigact;
17
18
19 /* Catch ctr-c */
20 void int_hand(int sig){
21   printf("\n%s: caught signal %d, exiting\n", ARGVZERO, sig);
22   exit(sig);
23 }
24
25 /*
26   badopt -- prints bad option error
27 */
28
29 void badopt(const int c){
30   fprintf(stderr, ""
31   "error: bad option: %c\n"
32   "Type \'%s -h\' for help.\n"
33   "", c, ARGVZERO);
34 }
35
36 void usage(){
37     fprintf(stderr, "usage: %s ", ARGVZERO);
38     fprintf(stderr, "-[hcfv] -[L <level>] -[j <device>] <filename>\n");
```

```
39      exit(0);
40  }  /* Usage */
41
42  /* ****************************************************************
43   * MAIN FUNCTION                                                  *
44   * ****************************************************************/
45  int main(int argc, char **argv) {
46    int opt;                              /* option character        */
47    int ret;
48    extern int optopt;                    /* getopt() stuff          */
49    extern char *optarg;                  /* getopt() stuff          */
50    extern int optind;                 /* getopt() stuff          */
51
52    /* Init options */
53    options.slevel.sanitize_file  = 0;
54    options.slevel.level          = 0;
55    options.verbose               = 0;
56    options.random_cdev           = 0;
57    options.journal_dev       = NULL;
58
59    /* Set signal handler */
60    sigact.sa_handler = int_hand;
61    sigemptyset(&sigact.sa_mask);
62    sigact.sa_flags = 0;
63    sigaction(SIGINT, &sigact, 0);
64
65    while ((opt = getopt(argc, argv, "hfcvL:j:")) != -1){
66      switch (opt){
67      case 'L': /* Sanitize level */
68        options.slevel.sanitize_file  = 1;
69        if((atoi(optarg) <= 0) || ((atoi(optarg) > 5)&&(atoi(optarg)!=9))){
70          fprintf(stderr, "%s: bad option: level range invalid.\n", ARGVZERO
                );
71          exit(BAD_USAGE);
72        }
73        options.slevel.level = atoi(optarg);
74        break;
75      case 'v':
76        options.verbose = 1;
77        break;
78      case 'f':
79        options.force = 1;
80        break;
81      case 'c':
82        options.random_cdev = 1;
83        break;
84      case 'j':
85        options.journal_dev = malloc (sizeof(char) * (strlen(optarg)+1));
86        strcpy(options.journal_dev,optarg);
87        break;
88      case 'h':
89        usage();
90        break;
91      default:
92        badopt(optopt);
93        break;
```

```
94      }
95    }
96
97  #ifdef OPTIONTEST
98    printf("options are:\n");
99    printf("sanitize_file     = %d\n", options.slevel.sanitize_file);
100   printf("sanitize level    = %d\n", options.slevel.level);
101   printf("verbose           = %d\n", options.verbose);
102   printf("force             = %d\n\n", options.force);
103   printf("random_cdev       = %d\n\n", options.random_cdev);
104   abort();
105 #endif
106
107   if((options.slevel.level > 2)&&(options.journal_dev==NULL)){
108     fprintf(stderr, "%s: bad option: level requires -j option.\n",
          ARGVZERO);
109     exit(BAD_USAGE);
110   }
111
112 #ifdef FILETEST
113   fprintf(stderr, "getopt() parsed %d args\n", optind - 1);
114 #endif
115   if (optind == argc){
116     /*show_copyright();*/
117     fprintf(stderr, "Filename missing!\nType \'%s -h\' for usage.\n",
          ARGVZERO);
118     exit(BAD_USAGE);
119   }
120
121   if((options.slevel.level > 2)&&( getuid()!= 0)){
122     fprintf(stderr, "%s: You need root access for this.\n",ARGVZERO);
123     exit(BAD_USAGE);
124   }
125
126   /* Wipe file. */
127   while (optind < argc){
128     printf("\nWiping file: %s",argv[optind]);
129     ret = file_wipefile(argv[optind++], &(options));
130     if((options.slevel.level>3)&&(ret != 17)){
131       printf("\nwipe failed! Erro %d",ret);
132     }else{
133       if((options.slevel.level==3)&&(ret!=9)){
134         printf("wipe failed! Erro %d",ret);
135       }else{
136         printf("\n[COMPLETE]\n");
137       }
138     }
139   }
140   printf("\n");
141
142   return ret;
143 }
```

Arquivo: ./src/file.c

```
1  #include <unistd.h>
2  #include <sys/types.h>
3  #include <stdio.h>
4  #include <stdbool.h>
5  #include <fcntl.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <errno.h>
9
10 #include "io.h"
11 #include "file.h"
12 #include "hitface.h"
13 #include "dev.h"
14 #include "journal.h"
15
16
17 /* ****************************************************************
18  * file_getdata - Stat the file and inits some values to filest_t   *
19  * struct.                                                          *
20  * ****************************************************************/
21 int file_getdata(char *fname, struct filest_t *f){
22   char *cptr;
23
24   /* Excluding . and .. notation. */
25   if((*fname == '.')&&(*(fname+1) == '.')){
26     printf("Do not use '.' or '..' notation.\n");
27     return FAILED;
28   }
29   if((*fname == '.')&&(strlen(fname) < 2)){
30     printf("Do not use '.' or '..' notation.\n");
31     return FAILED;
32   }
33
34   if((*fname == '.')&&(*(fname+1) == '/')&&(strlen(fname) < 3)){
35     printf("Do not use '.' or '..' notation.\n");
36     return FAILED;
37   }
38
39
40   /*File name and path.*/
41   strncpy(f->file_name, fname, PATH_MAX);
42   strncpy(f->pathname, fname, PATH_MAX);
43   f->file_name[sizeof(f->file_name)-1] = 0;
44   f->pathname[sizeof(f->pathname)-1] = 0;
45
46   /* Getting the pathname. */
47   cptr = strrchr(f->pathname, '/');
48   if (cptr == NULL || cptr >= ((f->pathname) + sizeof(f->pathname))){
49     cptr = f->pathname;
50   }else{
51     ++cptr;
52   }
53   /* Truncate the path.*/
54   *cptr = 0x00;
55
```

```
56    /* Take the case of dir = "./someway/" */
57    if(strlen(f->pathname) == strlen(f->file_name)){
58      cptr = (f->pathname + strlen(f->pathname) - 1);
59      *cptr = 0x00;
60      cptr = strrchr(f->pathname, '/');
61      if (cptr == NULL || cptr >= ((f->pathname) + sizeof(f->pathname))){
62        cptr = f->pathname;
63      }else{
64        ++cptr;
65      }
66      /* Truncate the path.*/
67      *cptr = 0x00;
68    }
69
70    /* Incase that the file is on the current directory. */
71    if(strlen(f->pathname)==0){
72      strcpy(f->pathname,"./");
73    }
74
75    if (lstat(f->file_name, &f->st)){
76      fprintf(stderr, "\r%s: cannot stat `%s': %s\n",
77        ARGVZERO, f->file_name, strerror(errno));
78      return FAILED;
79    }
80
81    /* Init file descriptor*/
82    f->fd = -1;
83
84    /* Init file buffer. */
85    f->buf = NULL;
86    f->bufsize = 0;
87
88    return 0;
89  }
90
91
92
93
94  /* ****************************************************************
95   * The core routine that do the work.  This overwrites the file of a *
96   * given fd.                                    *
97   * ************************************************************** */
98  int file_fillfile(struct filest_t *f, off_t fsize, struct opt_t *opt){
99    size_t nbytes;
100   off_t slim, soff;
101   char buf[4*SECTOR_SIZE];
102   int bad_block = 0;
103
104   /* Jump fd to the begin of the file. */
105   if (lseek (f->fd, 0, SEEK_SET) == -1){
106     fprintf(stderr, "\r%s: cannot seek to the begin of the file `%s': %s\n"
          , ARGVZERO, f->file_name, strerror(errno));
107     return FAILED;
108   }
109
110   if(opt->slevel.level > 4){
```

```
111       dev_getrandombuffer(&buf, 4*SECTOR_SIZE);
112       if(opt->slevel.level == 9){
113         slim = fsize + SECTOR_SIZE - 1 - (fsize-1) % SECTOR_SIZE;
114       }else{
115         slim = (24*SECTOR_SIZE>fsize) ? fsize : 24*SECTOR_SIZE;
116       }
117     }else{
118       dev_getzerobuffer(&buf, 4*SECTOR_SIZE);
119       slim = (8*SECTOR_SIZE>fsize) ? fsize : 8*SECTOR_SIZE;
120     }
121
122     nbytes = 0;
123     for(soff = 0 ; soff < slim ; soff += nbytes ){
124       nbytes = write(f->fd, buf, (size_t) sizeof(buf));
125       if(nbytes < 0){
126         /* Skiping BAD blocks in case of find them. */
127         if (errno == EIO && (0 <= slim - soff) && (soff | SECTOR_MASK) < slim
              ){
128           size_t soff_aux = (soff | SECTOR_MASK) + 1;
129           if (lseek (f->fd, soff_aux, SEEK_SET) != -1){
130             /* Arrange to skip this block. */
131             soff = soff_aux - nbytes;
132             bad_block = 1;
133             continue;
134           }
135           fprintf(stderr, "\r%s: cannot seek bad sector offset %d, `%s': %s\n
                ", ARGVZERO, (int)soff, f->file_name, strerror(errno));
136         }
137         return -1;
138       }
139     }
140
141     /*Make changes hit media. */
142     if (io_f_sync(f->fd, f->file_name) != 0){
143       return FAILED;
144     }
145
146     return bad_block;
147 }
148
149
150 /* ****************************************************************
151  * do_wipefd - The core routine that do the work. This overwrites the*
152  * of the given fd.                                 *
153  * **************************************************************** */
154 int file_do_wipefd (struct filest_t *f, struct opt_t *opt){
155
156
157     /* The wipe make sense only for regular files, directory or block special
           file. */
158     if( !(S_ISREG(f->st.st_mode)||S_ISDIR(f->st.st_mode)||S_ISBLK(f->st.
          st_mode)) ){
159       fprintf(stderr, "\r%s: invalid file type `%s': %s\n", ARGVZERO, f->
            file_name, strerror(errno));
160       return FAILED;
161     }
```

```
162
163   /* Do the work */
164   if(file_fillfile( f, f->st.st_size, opt) != 0){
165     fprintf(stderr, "\r%s: invalid file type `%s': %s\n", ARGVZERO, f->
          file_name, strerror(errno));
166     return FAILED;
167   }
168   return 0;
169 }
170
171 /* ***************************************************************
172  * file_wipefile - wipe the file                                 *
173  * *************************************************************** */
174 int file_wipefile (char *fname, struct opt_t *opt){
175   int ok, ret;
176   struct filest_t f;
177
178   /* Stat and get data of file */
179   if ((file_getdata(fname, &f)) == FAILED){
180     return FAILED;
181   }
182
183   ok = ret = 0;
184   /* Get bits of file mode that describe file type.*/
185   switch(f.st.st_mode & __S_IFMT){
186     case __S_IFREG: /* regular file */
187         /* Open file. */
188       f.fd = open (fname, O_WRONLY | O_NOCTTY);
189       if (f.fd < 0 && (errno == EACCES && opt->force) && chmod (fname,
            S_IWUSR) == 0){
190         f.fd = open (fname, O_WRONLY | O_NOCTTY );
191       }
192       if (f.fd < 0) {
193         fprintf(stderr, "\r%s: failed to open `%s': %s\n", ARGVZERO, fname,
              strerror(errno));
194         return FAILED;
195       }
196
197       /* 9 - Same of 3 more file overwrites the all file with pseudo-random
             values.*/
198       /* 5 - Same of 3 more file overwrites 3 blocks of the file with
            pseudo-random values. */
199       /* 4 - Same of 3 more file overwrites the first file block with zeros
            . */
200       if(opt->slevel.level > 3){
201         if(opt->verbose){
202           printf("\n\tLevel %d: starting ...",opt->slevel.level);
203         }
204         /* Wipe the content of a file. */
205         ok = file_do_wipefd (&f, opt);
206         if(ok == 0){
207           ret = 8;
208         }else{
209           printf("\nWarnning level %d failed.",opt->slevel.level);
210           ret = 0;
211         }
```

```
212         }
213
214         if((opt->slevel.level == 2)||(opt->slevel.level == 3)||((opt->slevel.
                level > 3)&&(ret != 0))){
215           if(opt->verbose){
216             printf("\n\tLevel 2: renaming [%d] ...",strlen(f.file_name));
217           }
218           /* 2 - Clean name, size and number of blocks. */
219           if(hitface_chg_name(f.fd,f.file_name, strlen(f.file_name), opt->
                random_cdev,opt->verbose) != 0){
220             fprintf(stderr, "\r%s: cannot wipe name `%s': %s\n", ARGVZERO, f.
                    file_name, strerror(errno));
221             ok = FAILED;
222           }
223           if(ok == 0){
224             ret = ret + 2;
225           }else{
226             printf("\nWarnning level 2 failed.");
227             ret = ret + 0;
228           }
229           if(opt->verbose){
230             printf(" [done]");
231           }
232         }
233
234         /* Fecha o arquivo. */
235         if (close (f.fd) != 0){
236           fprintf(stderr, "\r%s: failed to close `%s': %s\n", ARGVZERO, fname
                , strerror(errno));
237           return FAILED;
238         }
239
240         /* 1 - Unlink the file. */
241         ok = unlink (f.file_name);
242         if(ok == 0){
243           ret = ret + 1;
244         }else{
245           printf("\nWarnning level 1 failed.");
246           ret = ret + 0;
247         }
248
249         if(opt->slevel.level > 2){
250           /* Clear inode.*/
251           if(opt->verbose){
252             printf("\n\tLevel 3: cleaning [%s,%d]",opt->journal_dev,(int)f.st
                    .st_ino);
253           }
254           ok = journal_wipejournal(opt->journal_dev, (int)f.st.st_ino, opt);
255           if(ok > 0){
256             ret = ret + 4 + 2;
257           }else{
258             printf("\nWarnning! Level 3 failed.");
259             ret = ret + 0;
260           }
261           ok = 0;
262         }
```

```
263
264         break;
265       case __S_IFBLK: /* block device */
266         /*destroy_blkdev();*/
267         printf("\nNOT IMPLEMENTED! [FAILED]\n");
268         break;
269         case __S_IFDIR: /* directory */
270           printf("\nWIPE DIR NOT IMPLEMENTED! [FAILED]\n");
271           break;
272
273         default:
274           printf("\r%s:(file_wipefile): file type not supported - %x\n",
                ARGVZERO,(f.st.st_mode & __S_IFMT));
275           return FAILED;
276     }
277
278     return ret;
279 }
```

Arquivo: ./src/hitface.c

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <fcntl.h>
5 #include <sys/types.h>
6 #include <string.h>
7 #include <errno.h>
8
9 #include "hitface.h"
10 #include "mcT.h"
11 #include "io.h"
12 #include "file.h"
13 #include "dev.h"
14
15 /* ************************************************************
16  *     random_cdev -- return a safe low-ASCII char            *
17  ************************************************************ */
18 char random_cdev(void){
19   char buf;
20   do{
21     dev_getrandombuffer(&buf, sizeof(char));
22   }while((buf<48)||(buf>122));
23   return buf;
24 }
25
26
27 /* ************************************************************
28  *     rand_char -- return a safe low-ASCII char              *
29  ************************************************************ */
30 char rand_char(void){
31   int i;
32   const char charset[] = "0123456789
       abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_+=-;#.";
```

```
33    i = mcT_extract_number() % sizeof(charset) ;
34    return charset[i];
35  }
36
37
38  /* ************************************************************
39   *  rename_str -- fill a string with random chars             *
40    ************************************************************ */
41
42  void rename_str(char *str, const size_t len, int opt_random_cdev){
43    int i;
44    i=0;
45
46    if(opt_random_cdev){
47      while (i < (len-6)){
48        *(str+i) = random_cdev();
49        i++;
50      }
51      if(len > 6){
52        *(str + len-6) = '.';
53        *(str + len-5) = random_cdev();
54        *(str + len-2) = 0x00;
55      }else{
56        *(str + len-5) = '.';
57        *(str + len-2) = random_cdev();
58      }
59      *(str + len-4) = random_cdev();
60      *(str + len-3) = random_cdev();
61    }else{
62      while (i < (len-6)){
63        *(str + i) = rand_char();
64        i++;
65      }
66      if(len > 6){
67        *(str + len-6) = '.';
68        *(str + len-5) = rand_char();
69        *(str + len-2) = 0x00;
70      }else{
71        *(str + len-5) = '.';
72        *(str + len-2) = rand_char();
73      }
74      *(str + len-4) = rand_char();
75      *(str + len-3) = rand_char();
76    }
77    *(str + len-1) = 0x00;
78  }
79
80
81
82  /* ************************************************************
83   *  hitface_chg_name -- wipe name of a file                   *
84    ************************************************************ */
85  int hitface_chg_name(int fd, char *real_name, size_t nsize, int random_cdev
      ,int verbose){
86    int      i;
87    char     *cptr;
```

```
88    char      dest_name[PATH_MAX+1];
89    char      pathname[PATH_MAX+1];
90    size_t      len, pathlen;
91
92
93    if (nsize == 0 || nsize > NAME_MAX){
94      nsize = NAME_MAX;
95    }
96    if (nsize > 0){
97      ++nsize;
98    }
99
100   strncpy(dest_name, real_name, nsize);
101   dest_name[sizeof(dest_name)-1] = 0x00;
102
103   cptr = strrchr(dest_name, '/');
104
105   if (cptr == NULL || cptr >= (dest_name + sizeof(dest_name))){
106     cptr = dest_name;
107   }else{
108     ++cptr;
109   }
110
111   /* Truncate the path.*/
112   *cptr = 0x00;
113
114   /* Get the length of the path. */
115   pathlen = strnlen(dest_name, sizeof(dest_name));
116   strcpy(pathname,dest_name);
117
118   /* Filename has to be smaller than PATH_MAX. */
119   len = pathlen + nsize;
120   if (len > PATH_MAX){
121     nsize -= len - PATH_MAX;
122   }
123
124   /* Get a random filename */
125   i=0;
126   while (!i){
127     rename_str(cptr, (nsize - pathlen+1), random_cdev);
128     /* Check if the new name exist - unistd.h*/
129     i = access(dest_name, F_OK);
130   }
131   if (rename(real_name, dest_name) != 0){
132     fprintf(stderr, "\r%s: cannot rename `%s': %s\n", ARGVZERO, real_name,
          strerror(errno));
133     return FAILED;
134   }
135
136   strncpy(real_name, dest_name, strnlen(dest_name, sizeof(dest_name)));
137   real_name[PATH_MAX] = 0;
138
139   if(strlen(pathname)==0){
140     strcpy(pathname,"./");
141   }
142   /* now try to commit the rename to storage */
```

```
143   if (io_dir_sync(fd, pathname)){
144     return FAILED;
145   }
146
147   if (fd == -1){
148     /* if the object is itself a dir, sync it also */
149     if (io_dir_sync(fd, real_name)){
150       return FAILED;
151     }
152   }
153   /* Save new name. */
154   strcpy(real_name, dest_name);
155   if(verbose){
156     printf(" [%s]",dest_name);
157   }
158   return 0;
159 }
```

Arquivo: ./src/dev.c

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <sys/stat.h>
4 #include <fcntl.h>
5 #include <errno.h>
6 #include <string.h>
7 #include <stdlib.h> /*exit*/
8 #include "dev.h"
9
10 /*  ****************************************************************
11  * dev_rand_init -- inits the entropy source file descriptor.      *
12  **************************************************************** */
13 int dev_rand_init(int *entropyfd, char *entropy_name){
14   /* Try /dev/urandom first; if that fails, try /dev/random */
15   if ((*entropyfd = open("/dev/urandom", O_RDONLY)) < 0){
16     if ((*entropyfd = open("/dev/random", O_RDONLY)) < 0){
17       fprintf(stderr, "\r%s: cannot open entropy source: %s\n", ARGVZERO,
             strerror(errno));
18       exit(1);
19     }else{
20       strncpy(entropy_name, "/dev/random", sizeof(entropy_name));
21       fprintf(stderr, "\r%s: warning: cannot open /dev/urandom, "
22           "using /dev/random instead\n", ARGVZERO);
23     }
24   }else{
25     strncpy(entropy_name, "/dev/urandom", sizeof(entropy_name));
26   }
27   return 0;
28 }
29
30 /* ****************************************************************
31  * dev_zeros_init -- inits zero source file descriptor.      *
32  ****************************************************************/
33 int dev_zeros_init(int *fd){
```

```
34    /* Try /dev/zeros first; if that fails, try /dev/random */
35    if ((*fd = open("/dev/zero", O_RDONLY)) < 0){
36      fprintf(stderr, "\r%s: cannot open zero source: %s\n", ARGVZERO,
          strerror(errno));
37      exit(1);
38    }
39    return 0;
40  }
41  /* ****************************************************************
42   * dev_getrandombuffer - gets buffer of size "size" and fill it with *
43   * random values.                                                   *
44   **************************************************************** */
45  ssize_t dev_getrandombuffer(void *buf, ssize_t size){
46    ssize_t ret = 0;
47    char *source_name = malloc(sizeof(char)*15);
48    int fd, retry = 0;
49
50    if(dev_rand_init(&fd, source_name) != 0){
51      fprintf(stderr, "\r%s: cannot init rand source.\n", ARGVZERO);
52      return FAILED;
53    }
54    /* Get random buf. */
55    while((ret < size)&&(retry < 5)){
56      ret = ret + read( fd, buf + ret, size-ret);
57      if(ret < 0){
58        if((errno == EINTR) || (errno == EIO)){
59          retry++;
60        }else{
61          retry = 5;
62        }
63      }
64    }
65
66    if(retry >= 5){
67      fprintf(stderr, "\r%s: randombuffer: cannot read from source fd!\n",
          ARGVZERO);
68      if(close(fd)){
69        fprintf(stderr, "\r%s: cannot close fd", ARGVZERO);
70      }
71      return FAILED;
72    }
73
74    if(close(fd)){
75      fprintf(stderr, "\r%s: cannot close fd", ARGVZERO);
76      return FAILED;
77    }
78    return (ret);
79  }
80
81   /* ****************************************************************
82   * dev_getzerobuffer - gets a buffer of size "size" and fill it with  *
83   * zeros.                                                           *
84   **************************************************************** */
85  ssize_t dev_getzerobuffer(void *buf, ssize_t size){
86    ssize_t ret = 0;
87    int fd, retry = 0;
```

```
88
89    if(dev_zeros_init(&fd) != 0){
90      fprintf(stderr, "\r%s: cannot init rand source.\n", ARGVZERO);
91      return FAILED;
92    }
93    /* Get zero buf. */
94    while((ret < size)&&(retry < 5)){
95      ret = ret + read( fd, buf + ret, size-ret);
96      if(ret < 0){
97        if((errno == EINTR) || (errno == EIO)){
98          retry++;
99        }else{
100         retry = 5;
101       }
102     }
103   }
104
105   if(retry >= 5){
106     fprintf(stderr, "\r%s: zerobuffer: cannot read from source fd!\n",
          ARGVZERO);
107     if(close(fd)){
108       fprintf(stderr, "\r%s: cannot close fd", ARGVZERO);
109     }
110     return FAILED;
111   }
112
113   if(close(fd)){
114     fprintf(stderr, "\r%s: cannot close fd", ARGVZERO);
115     return FAILED;
116   }
117   return (ret);
118 }
```

Arquivo: ./src/io.c

```
1  /* fsync */
2  #include <unistd.h>
3  /* Dirfd e diropen */
4  #include <sys/types.h>
5  #include <dirent.h>
6  #include <fcntl.h>
7
8  #include <errno.h>
9  #include <string.h>
10 #include <stdio.h>
11
12 #include "io.h"
13 #include "config.h"
14
15
16
17 /* ************************************************************
18  * io_dir_sync - Sync directory changes to the disk.        *
19  * ************************************************************ */
```

```
20 int io_dir_sync(int ffd, char *name) {
21 #ifdef HAVE_DIRFD
22   int dfd;
23   DIR *dir;
24
25   /* sync the directory */
26   if ((dir = opendir(name)) == NULL){
27     fprintf(stderr, "\r%s: cannot open directory `%s': %s\n", ARGVZERO,
         name, strerror(errno));
28     return FAILED;
29   }
30   if ((dfd = dirfd(dir)) < 0){
31     fprintf(stderr, "\r%s: dirfd() failed for `%s': %s\n", ARGVZERO, name,
         strerror(errno));
32     return FAILED;
33   }
34
35   io_f_sync(dfd, name);
36
37   if (closedir(dir)){
38     fprintf(stderr, "\r%s: closedir failed for `%s': %s\n", ARGVZERO, name,
         strerror(errno));
39     return FAILED;
40   }
41 #endif
42
43   /* fsync() the file, in some OS */
44   if (ffd > -1){
45     if(io_f_sync(ffd, name) != 0){
46 #ifdef HAVE_FCNTL
47       if (fcntl(ffd, F_SETFL, O_SYNC) == -1){
48         fprintf(stderr, "\r%s: cannot set synchronis writes `%s': %s\n",
             ARGVZERO, name, strerror(errno));
49       }
50       return FAILED;
51 #endif
52     }
53   }
54   return 0;
55 }
56
57
58 int io_f_sync(int fd, char *name){
59
60   #if HAVE_FDATASYNC
61     if (fdatasync (fd) == 0){
62       return 0;
63     }else{
64       if(!(errno == EINVAL || errno == EBADF || errno == EISDIR)){
65         fprintf(stderr, "\r%s: fdatasync failed `%s': %s\n",ARGVZERO, name,
              strerror(errno));
66         return FAILED;
67       }
68     }
69   #endif
70
```

```
71    if (fsync (fd) == 0){
72      return 0;
73    }else{
74      if(!(errno == EINVAL || errno == EBADF || errno == EISDIR)){
75        fprintf(stderr, "\r%s: fdatasync failed `%s': %s\n",ARGVZERO, name,
              strerror(errno));
76        return FAILED;
77      }
78    }
79    /* It will not return as long as there is data which has not been written
          to the device. */
80    sync ();
81    return 0;
82  }
```

Arquivo: ./src/inode.c

```
1  #include "inode.h"
2  /* ************************************************************
3   * print_buffer -- print buffer as a hexdump. Used for debug.   *
4   * ************************************************************ */
5  void print_buffer2(int buflen, void *ptr){
6    unsigned char *buf = (unsigned char*) ptr;
7    int i,j;
8
9    for(i=0 ; i<buflen ; i = i + 16){
10     printf("%06x: ",i);
11     for(j=0; j<16;j++){
12       if(i+j < buflen)
13         printf("%02x",buf[i+j]);
14       else
15         printf(" | ");
16       if((i+j+1)%2 == 0){
17         printf(" ");
18       }
19     }
20     printf(" ");
21     for(j=0; j<16;j++){
22       if(i+j<buflen){
23         printf("%c", isprint(buf[i+j]) ? buf[i+j] : '.' );
24       }
25     }
26     printf("\n");
27   }
28 }
29
30 /* ************************************************************
31  * calc_inodetb_seek -- calculate the seek to inode table.      *
32  * ************************************************************ */
33 off_t calc_inodetb_seek(off_t block_size, off_t group){
34
35    return ((0x400<<block_size) + (0x20*group));
36 }
37
```

```
38 /* ************************************************************
39  * calc_inode_seek -- calculate the seek to inode.          *
40  * ************************************************************ */
41 off_t calc_inode_seek(off_t block_size, off_t inode_table, off_t inode_size
       , off_t inode_in_group){
42
43   return (((0x400<<block_size)*(inode_table)) + (inode_size*inode_in_group)
         );
44 }
45
46 /* ************************************************************
47  * dev_finode_clean -- clear a inode.                       *
48  * ************************************************************ */
49 int dev_finode_clean(int *fd, struct frs_superblock fsb, int finode){
50   unsigned int pos_finode_in_group;     /* Position of the finode in the
         group.*/
51   unsigned int finode_group;          /* Number of the group of the finode.*/
52   unsigned int gd_inode_start;        /* Start of the group descriptor. */
53   void *ibuffer;
54   int ret = 0;
55
56   ibuffer = malloc(fsb.fs_inode_size+1);
57
58   /* Number of the inode in the group.*/
59   pos_finode_in_group = (finode - 0x01)%(fsb.inodes_per_group);
60   /* Getting the group of which the inode belongs.*/
61   finode_group = (finode - 0x01 - pos_finode_in_group)/(fsb.
         inodes_per_group);
62
63   /* Getting fildes to the inode table*/
64   if(lseek(*fd,(off_t)(calc_inodetb_seek(fsb.fs_block_size,finode_group) +
         0x08), SEEK_SET) == (off_t)(-1)){
65     fprintf(stderr, "\r%s: cannot seek to descriptor block: %s\n", ARGVZERO
           , strerror(errno));
66     printf("Value of fs_block_size: %du\n",fsb.fs_block_size);
67     exit(1);
68   }
69   frs_byte_read(fd,&(gd_inode_start),4);
70
71   /* Inode table read. */
72   if(pos_finode_in_group == 0){
73     if(lseek(*fd, (off_t)(calc_inode_seek(fsb.fs_block_size, gd_inode_start
           , fsb.fs_inode_size, pos_finode_in_group) + fsb.fs_inode_size),
          SEEK_SET) == (off_t)(-1)){
74       fprintf(stderr, "\r%s: cannot seek file inode: %s\n", ARGVZERO,
             strerror(errno));
75       printf("Value of fs_block_size: %du\n",fsb.fs_block_size);
76       exit(1);
77     }
78     ret = read(*fd, ibuffer, fsb.fs_inode_size);
79     if(lseek(*fd, (off_t)(calc_inode_seek(fsb.fs_block_size, gd_inode_start
           , fsb.fs_inode_size, pos_finode_in_group)), SEEK_SET) == (off_t)(-1)
           ){
80       fprintf(stderr, "\r%s: cannot seek file inode: %s\n", ARGVZERO,
             strerror(errno));
81       printf("Value of fs_block_size: %du\n",fsb.fs_block_size);
```

```
82        exit(1);
83      }
84    }else{
85      if(lseek(*fd, (off_t)(calc_inode_seek(fsb.fs_block_size, gd_inode_start
            , fsb.fs_inode_size, pos_finode_in_group) - fsb.fs_inode_size),
           SEEK_SET) == (off_t)(-1)){
86        fprintf(stderr, "\r%s: cannot seek file inode: %s\n", ARGVZERO,
              strerror(errno));
87        printf("Value of fs_block_size: %du\n",fsb.fs_block_size);
88        exit(1);
89      }
90      ret = read(*fd, ibuffer, fsb.fs_inode_size);
91    }
92
93    ret = ret + write(*fd, ibuffer, (fsb.fs_inode_size));
94    if(ret != 2*(fsb.fs_inode_size)){
95      printf("ERROR: Problems cleaning qthe INODE.\nRET: %d",ret);
96      return -1;
97    }
98
99    return ret;
100 }
```

Arquivo: ./src/journal.c

```
1  #include "journal.h"
2  #include "inode.h"
3  #include "dev.h"
4  #include "io.h"
5
6  /* ****************************************************************
7   * print_buffer -- print buffer as a hexdump.                     *
8   * ****************************************************************/
9  void print_buffer(int buflen, void *ptr){
10   unsigned char *buf = (unsigned char*) ptr;
11   int i,j;
12
13   for(i=0 ; i<buflen ; i = i + 16){
14     printf("%06x: ",i);
15     for(j=0; j<16;j++){
16       if(i+j < buflen)
17         printf("%02x",buf[i+j]);
18       else
19         printf(" | ");
20       if((i+j+1)%2 == 0){
21         printf(" ");
22       }
23     }
24     printf(" ");
25     for(j=0; j<16;j++){
26       if(i+j<buflen){
27         printf("%c", isprint(buf[i+j]) ? buf[i+j] : '.' );
28       }
29     }
```

```
30      printf("\n");
31    }
32  }
33
34  /* ****************************************************************
35   * frs_bigendian_byte_read -- read from fildes 1,2 or 4 bytes those *
36   * are in bigendian order (JOURNAL USES).                           *
37   **************************************************************** */
38  int frs_bigendian_byte_read(int *fd, unsigned int *buf, size_t nsize) {
39    int i;
40    int aux;
41    int var_toReturn = 0;
42    assert (*fd != 0);
43
44    (*buf) = (*buf) & ~(*buf);
45    for(i=0;i<nsize;i++){
46      aux = 0x00000000;
47      var_toReturn = var_toReturn + read(*fd, &aux, 1);
48      if(nsize == 4){
49        *buf = (aux<<((3-i)*8)) | *buf;
50      }else{
51        if(nsize == 2){
52          *buf = (aux<<((1-i)*8)) | *buf;
53        }else{
54          assert(nsize == 1);
55          *buf = (aux | *buf);
56        }
57      }
58    }
59    return var_toReturn;
60  }
61
62  /* ****************************************************************
63   * journal_readsb_journal -- get info about the journal structure.  *
64   * ************************************************************** */
65  void journal_readsb_journal(int *fd, struct journal_sb *jsb, int verbose){
66    unsigned int aux;
67
68    /* Process progress.*/
69    if(verbose){
70      printf(".");
71    }
72
73    /* Journal signature: bytes 0-3 */
74    frs_bigendian_byte_read(fd,&(jsb->signature),4);
75    /* Block type (1 - Descriptor,2 Commit,3 SB v1,4
76     * SB v2,5 Revoke): bytes 4-7 */
77    frs_bigendian_byte_read(fd,&(jsb->block_type),4);
78    /* Sequencial number: bytes 8-11 */
79    frs_bigendian_byte_read(fd,&(jsb->seq_num),4);
80    /* Journal block size: bytes 12-15 */
81    frs_bigendian_byte_read(fd,&(jsb->block_size),4);
82    /* Number of journal blocks: bytes 16-19 */
83    frs_bigendian_byte_read(fd,&(jsb->qtd_blocks),4);
84    /* Journal block where the journal actually starts: bytes 20-23 */
85    frs_bigendian_byte_read(fd,&(jsb->jstart_block),4);
```

```
86    /* Sequence number of first transaction: bytes 24-27 */
87    frs_bigendian_byte_read(fd,&(jsb->seq_tr),4);
88    /* Journal block of first transaction: bytes 28-31 */
89    frs_bigendian_byte_read(fd,&(jsb->bfst_tr),4);
90    /* Error codes */
91    frs_bigendian_byte_read(fd,&(jsb->erron_code),4);
92    /* Compatible fealures */
93    frs_bigendian_byte_read(fd,&(jsb->cfealure),4);
94    /* Imcompatible fealures */
95    frs_bigendian_byte_read(fd,&(jsb->ifealure),4);
96    /* Read only fealures */
97    frs_bigendian_byte_read(fd,&(jsb->rofealure),4);
98
99    /* Journal UUID. */
100   frs_bigendian_byte_read(fd,&(jsb->uuid),4);
101   (aux) = (aux) & ~(aux);
102   aux = (jsb->uuid)<<4;
103   frs_bigendian_byte_read(fd,&(jsb->uuid),4);
104   aux = aux | jsb->uuid;
105   aux = aux<<4;
106   frs_bigendian_byte_read(fd,&(jsb->uuid),4);
107   aux = aux | jsb->uuid;
108   aux = aux<<4;
109   frs_bigendian_byte_read(fd,&(jsb->uuid),4);
110   jsb->uuid = aux | jsb->uuid;
111
112   /* Number of file system using the journal. */
113   frs_bigendian_byte_read(fd,&(jsb->fsqtd),4);
114   /* Location of super block copy. */
115   frs_bigendian_byte_read(fd,&(jsb->sbcopy),4);
116   /* Max number of journal blocks per transaction. */
117   frs_bigendian_byte_read(fd,&(jsb->max_jbpt),4);
118   /* Max number of file system blocks per transaction. */
119   frs_bigendian_byte_read(fd,&(jsb->max_fsbpt),4);
120
121   /* Process progress.*/
122   if(verbose){
123     printf(".");
124   }
125 }
126
127
128 /* ****************************************************************
129  * uxread --- read from fildes 1,2 or 4 bytes (X = 1,2 or 4).     *
130  **************************************************************** */
131
132 int frs_byte_read(int *fd, unsigned int *buf, size_t nsize) {
133   int i;
134   int aux;
135   int var_toReturn = 0;
136   assert (*fd != 0);
137
138   (*buf) = (*buf) & ~(*buf);
139   for(i=0;i<nsize;i++){
140     aux = 0x00000000;
141     var_toReturn = var_toReturn + read(*fd, &aux, 1);
```

```
142     *buf = (aux<<(i*8)) | *buf;
143   }
144   return var_toReturn;
145 }
146 
147 /* ****************************************************************
148  * dev_ext3_init -- init file descriptor.                       *
149  **************************************************************** */
150 int dev_ext3_init(int *fd, const char *dev){
151   /* Try open device.*/
152   if ((*fd = open(dev, O_RDWR , S_IRUSR | S_IWUSR )) < 0){
153     fprintf(stderr, "\r%s:journal: cannot open device: %s\n", ARGVZERO,
          strerror(errno));
154     exit(-1);
155   }
156   return 0;
157 }
158 
159 /* ****************************************************************
160  * jornal_get_inode_data -- give a inode number, it fills the inode *
161  *              structure.                                      *
162  **************************************************************** */
163 void jornal_get_inode_data(int *fd, unsigned int inode_number,struct
    frs_superblock *fsb, struct frs_inode *fi, int verbose){
164   unsigned int inodenumb_in_group;        /* Position of the inode in the
       group.*/
165   unsigned int inode_group;       /* Number of the group of the inode.*/
166   unsigned int gd_inode_start;      /* Start of the group descriptor. */
167 
168   /* Process progress.*/
169   if(verbose){
170     printf(".");
171   }
172   /* Number of the inode in the group.*/
173   inodenumb_in_group = (inode_number - 0x01)%(fsb->inodes_per_group);
174   /* Getting the group of the inode.*/
175   inode_group = (inode_number - 0x01 - inodenumb_in_group)/(fsb->
       inodes_per_group);
176 
177   /* Group descriptor read. Should be on the block after the block
178    * where superblock is stored. It depends of the file system block
179    * size and the group witch the inode belongs. */
180   if(lseek(*fd,(off_t)(calc_inodetb_seek(fsb->fs_block_size,inode_group) +
       0x08), SEEK_SET) == (off_t)(-1)){
181     fprintf(stderr, "\r%s: cannot seek to descriptor block: %s\n", ARGVZERO
         , strerror(errno));
182     printf("Value of fs_block_size: %du\n",fsb->fs_block_size);
183     exit(1);
184   }
185 
186   frs_byte_read(fd,&(gd_inode_start),4);
187 
188   /* Inode table read. */
189   printf("ITableA.%x",(int) gd_inode_start);
```

```
190   if(lseek(*fd, (off_t)(calc_inode_seek(fsb->fs_block_size, gd_inode_start,
          fsb->fs_inode_size, inodenumb_in_group) + 0x08), SEEK_SET) == (off_t)
          (-1)){
191     fprintf(stderr, "\r%s: cannot seek to descriptor block: %s\n", ARGVZERO
            , strerror(errno));
192     printf("Value of fs_block_size: %du\n",fsb->fs_block_size);
193     exit(1);
194   }
195   /*The four time values are each stored as the number of seconds since
          January 1, 1970 UTC.*/
196   /* Access Time: bytes 8-11 */
197   frs_byte_read(fd,&(fi->a_time),4);
198   /* Change Time: bytes 12-15 */
199   frs_byte_read(fd,&(fi->c_time),4);
200   /* Modification time: bytes 16-19 */
201   frs_byte_read(fd,&(fi->m_time),4);
202   /* Deletion time: bytes 20-23 */
203   frs_byte_read(fd,&(fi->d_time),4);
204
205   if(lseek(*fd, 16, SEEK_CUR) == -1){
206     fprintf(stderr, "\r%s: failure on seeking inode block_pointers: %s\n",
          ARGVZERO, strerror(errno));
207     exit(1);
208   }
209
210   /* 12 direct block pointers: bytes 40-87 */
211   frs_byte_read(fd,&(fi->block_pointer1),4);
212   frs_byte_read(fd,&(fi->block_pointer2),4);
213   frs_byte_read(fd,&(fi->block_pointer3),4);
214   frs_byte_read(fd,&(fi->block_pointer4),4);
215   frs_byte_read(fd,&(fi->block_pointer5),4);
216   frs_byte_read(fd,&(fi->block_pointer6),4);
217   frs_byte_read(fd,&(fi->block_pointer7),4);
218   frs_byte_read(fd,&(fi->block_pointer8),4);
219   frs_byte_read(fd,&(fi->block_pointer9),4);
220   frs_byte_read(fd,&(fi->block_pointer10),4);
221   frs_byte_read(fd,&(fi->block_pointer11),4);
222   frs_byte_read(fd,&(fi->block_pointer12),4);
223   /* 1 single indirect block pointer: bytes 88-91 */
224   frs_byte_read(fd,&(fi->s_indirect_pointer),4);
225   /* 1 double indirect block pointer: bytes 92-95  */
226   frs_byte_read(fd,&(fi->d_indirect_pointer),4);
227   /* 1 triple indirect block pointer: bytes 96-99 */
228   frs_byte_read(fd,&(fi->t_indirect_pointer),4);
229
230   if(verbose){
231     printf(".");
232   }
233
234 }
235
236 /* ***************************************************************
237  * dev_ext3_track - walk the fd until the bytes of the journal file.*
238  *************************************************************** */
239 int dev_ext3_track(int *fd, struct frs_superblock *fsb, struct frs_inode *
      fi, int verbose){
```

```
240
241   if(verbose){
242     printf("\n\tGetting journal inode information .");
243   }
244
245   /* SEEK_SET set to offset bytes the file offset.*/
246   if(lseek(*fd, 1024, SEEK_SET) != 1024){
247     fprintf(stderr, "\r%s: cannot seek on device: %s\n", ARGVZERO, strerror
          (errno));
248     exit(1);
249   }
250
251   /* Superblock read.*/
252   frs_byte_read(fd,&(fsb->total_inodes_number),4);
253   frs_byte_read(fd,&(fsb->total_blocks_number),4);
254   frs_byte_read(fd,&(fsb->total_blocks_reserved),4);
255   if(lseek(*fd, 8, SEEK_CUR) != 1044){
256     fprintf(stderr, "\r%s: failure on seeking to start_group descriptor
          bytes: %s\n", ARGVZERO, strerror(errno));
257     exit(1);
258   }
259   frs_byte_read(fd,&(fsb->start_block_group_zero),4);
260   frs_byte_read(fd,&(fsb->fs_block_size),4);
261   frs_byte_read(fd,&(fsb->fs_fragment_size),4);
262   frs_byte_read(fd,&(fsb->blocks_per_group),4);
263   frs_byte_read(fd,&(fsb->fragments_per_group),4);
264   frs_byte_read(fd,&(fsb->inodes_per_group),4);
265
266   /* SEEK_CUR  set to its current location plus offset the file offset. */
267   if(lseek(*fd, 12, SEEK_CUR) != 1080){
268     fprintf(stderr, "\r%s: failure on seeking to fs_signature bytes: %s\n",
          ARGVZERO, strerror(errno));
269     exit(1);
270   }
271   frs_byte_read(fd,&(fsb->fs_signature),2);
272   if(fsb->fs_signature != 0xef53){
273     fprintf(stderr, "\r%s: Signature of partition does not match: %s\n",
          ARGVZERO, strerror(errno));
274     printf("Signature stored: %x\n",(fsb->fs_signature));
275     exit(1);
276   }
277
278   /* SEEK_CUR  set to its current location plus offset the file offset. */
279   if(lseek(*fd, 14, SEEK_CUR) != 1096){
280     fprintf(stderr, "\r%s: failure on seeking to fs_os_creator bytes: %s\n"
          , ARGVZERO, strerror(errno));
281     exit(1);
282   }
283   frs_byte_read(fd,&(fsb->fs_os_creator),4);
284   if(lseek(*fd, 12, SEEK_CUR) != 1112){
285     fprintf(stderr, "\r%s: failure on seeking to fs_inode_size bytes: %s\n"
          , ARGVZERO, strerror(errno));
286     exit(1);
287   }
288   frs_byte_read(fd,&(fsb->fs_inode_size),2);
289   if(lseek(*fd, 110, SEEK_CUR) != 1224){
```

```
290       fprintf(stderr, "\r%s: failure on seeking to fs_inode_size bytes: %s\n"
              , ARGVZERO, strerror(errno));
291       exit(1);
292     }
293     frs_byte_read(fd,&(fsb->bitmap_algoritm),4);
294     if(lseek(*fd, 20, SEEK_CUR) != 1248){
295       fprintf(stderr, "\r%s: failure on seeking to fs_inode_size bytes: %s\n"
              , ARGVZERO, strerror(errno));
296       exit(1);
297     }
298     frs_byte_read(fd,&(fsb->fs_journal_inode),4);
299     if(fsb->fs_journal_inode != 8){
300       fprintf(stderr, "\r%s: Jornal inode diferent. Not suported: %d\n",
             ARGVZERO, fsb->fs_journal_inode);
301       exit(1);
302     }
303     frs_byte_read(fd,&(fsb->fs_journal_device),4);
304
305
306
307     /* Gating Inode data. */
308     jornal_get_inode_data(fd, fsb->fs_journal_inode, fsb, fi, verbose);
309
310     return 0;
311 }
312
313
314 /* ****************************************************************
315  * dev_ext3_overwrite - return the amount of bytes overwritten on   *
316  *            the journal                                           *
317  **************************************************************** */
318 ssize_t dev_ext3_overwrite(int *fd, struct frs_superblock *fsb, struct
      frs_inode *fi, int verbose){
319     unsigned char *buf;
320     unsigned long long int long_seek;
321     unsigned int devide;
322     ssize_t nbytes, bsize, ret = 0;
323     int i,j,k,l, aux;
324     struct journal_sb jsb;
325     unsigned int jsbcleaner;
326     unsigned int *pull_indirect_pointer;
327     unsigned int *pull_direct_pointer;
328     /*Var to map journal blocks*/
329     pull_direct_pointer = (unsigned int *) malloc(sizeof(unsigned int) *
          32768);
330     *pull_direct_pointer = 0x00;
331
332     bsize = 0x400<<fsb->fs_block_size;
333     long_seek = (unsigned long long int) (bsize * (*(&(fi->block_pointer1))))
          ;
334     if (long_seek > 2147483647){
335       devide = (unsigned int) long_seek%2147483647;
336       long_seek = (unsigned long long int) long_seek/2147483647;
337       if(lseek64(*fd, (off_t) 2147483647, SEEK_SET) == -1){
338         fprintf(stderr, "\r%s: failure on seeking inode first block_pointers
              FAR01: %s\n", ARGVZERO, strerror(errno));
```

```
339         exit(1);
340       }
341       long_seek= long_seek-1;
342       while(long_seek > 1){
343         if(lseek64(*fd, (off_t) 2147483647, SEEK_CUR) == -1){
344           fprintf(stderr, "\r%s: failure on seeking inode first
                block_pointers FAR02: %s\n", ARGVZERO, strerror(errno));
345           exit(1);
346         }
347         long_seek= long_seek-1;
348       }
349       if(lseek64(*fd, (off_t) devide, SEEK_CUR) == -1){
350         fprintf(stderr, "\r%s: failure on seeking inode first block_pointers
              FAR03: %s\n", ARGVZERO, strerror(errno));
351         exit(1);
352       }
353     }else{
354       if(lseek(*fd, (off_t)(bsize * (*(&(fi->block_pointer1)))), SEEK_SET) ==
            -1){
355         fprintf(stderr, "\r%s: failure on seeking inode first block_pointers:
              %s\n", ARGVZERO, strerror(errno));
356         exit(1);
357       }
358     }
359     /* Journal Super Block Read. */
360     journal_readsb_journal(fd, &jsb,verbose);
361
362     long_seek = (unsigned long long int) (bsize * (*(&(fi->block_pointer1))))
          ;
363     if (long_seek > 2147483647){
364       devide = (unsigned int) long_seek%2147483647;
365       long_seek = (unsigned long long int) long_seek/2147483647;
366       if(lseek64(*fd, (off_t) 2147483647, SEEK_SET) == -1){
367         fprintf(stderr, "\r%s: failure on seeking inode first block_pointers
              FAR01: %s\n", ARGVZERO, strerror(errno));
368         exit(1);
369       }
370       long_seek= long_seek-1;
371       while(long_seek > 1){
372         if(lseek64(*fd, (off_t) 2147483647, SEEK_CUR) == -1){
373           fprintf(stderr, "\r%s: failure on seeking inode first
                block_pointers FAR02: %s\n", ARGVZERO, strerror(errno));
374           exit(1);
375         }
376         long_seek= long_seek-1;
377       }
378       if(lseek64(*fd,(off_t) devide, SEEK_CUR) == -1){
379         fprintf(stderr, "\r%s: failure on seeking inode first block_pointers
              FAR03: %s\n", ARGVZERO, strerror(errno));
380         exit(1);
381       }
382     }else{
383       if(lseek(*fd, (off_t)(bsize * (*(&(fi->block_pointer1)))), SEEK_SET) ==
            -1){
384         fprintf(stderr, "\r%s: failure on seeking inode first block_pointers:
              %s\n", ARGVZERO, strerror(errno));
```

```
385         exit(1);
386       }
387     }
388     if(lseek64(*fd, 16, SEEK_CUR) == -1){
389       fprintf(stderr, "\r%s: failure on seeking 0x10 DATA BLOCKS: %s\n",
              ARGVZERO, strerror(errno));
390       exit(1);
391     }
392
393     /* Inserting data of a cleaner journal.*/
394     jsbcleaner = 0x00100000;
395     ret = ret + write(*fd, (void *) &jsbcleaner, 4);
396     jsbcleaner = 0x01000000;
397     ret = ret + write(*fd, (void *) &jsbcleaner, 4);
398     ret = ret + write(*fd, (void *) &jsbcleaner, 4);
399     jsbcleaner = 0x00000000;
400     ret = ret + write(*fd, (void *) &jsbcleaner, 4);
401
402     /* Buffer to overwrite*/
403     buf = (unsigned char *) malloc(bsize);
404     dev_getzerobuffer(buf, bsize);
405
406     /* Jump to where the journal entries start. */
407     if(((bsize * (fi->block_pointer1)) + (jsb.jstart_block * jsb.block_size))
            != (bsize * (fi->block_pointer2))){
408       printf("\nWARNING: Start Journal Block != JournalInodeblock_pointer2.\n
              ");
409       if(lseek64(*fd, (off_t)((bsize * (fi->block_pointer1)) + (jsb.
              jstart_block * jsb.block_size)), SEEK_SET) == -1){
410         fprintf(stderr, "\r%s: failure on seeking to start entry inode block:
                %s\n", ARGVZERO, strerror(errno));
411         exit(-1);
412       }
413       ret = ret + write(*fd, buf, bsize);
414     }
415
416     for(i=1;i<12;i++){
417       if(lseek64(*fd, (off_t)(bsize * (*(&(fi->block_pointer1)+i))), SEEK_SET
              ) == -1){
418         fprintf(stderr, "\r%s: failure on seeking inode block_pointers: %s\n"
                , ARGVZERO, strerror(errno));
419         exit(-1);
420       }
421       ret = ret + write(*fd, buf, bsize);
422       if(verbose){
423         printf(".");
424       }
425     }
426     nbytes = bsize;
427     if(fi->s_indirect_pointer != 0x00000000){
428       if(lseek64(*fd, (off_t)(bsize * (fi->s_indirect_pointer)), SEEK_SET) ==
              -1){
429         fprintf(stderr, "\r%s: failure on seeking inode indirect
              block_pointers: %s\n", ARGVZERO, strerror(errno));
430         exit(-1);
431       }
```

```
432       /* Carregando endereÃ§os dos blocos diretos por meio do endereÃ§o de
              endireÃ§Ã£o simples. */
433       if(verbose){
434         printf(".SI.");
435       }
436
437       frs_byte_read(fd,pull_direct_pointer,4);
438       i = 1;
439       while((i < nbytes/4)&&(*(pull_direct_pointer+i-1) != 0x00)){
440         frs_byte_read(fd,(pull_direct_pointer+i),4);
441         i++;
442       }
443       if(verbose){
444         printf("%d.",i);
445       }
446       *(pull_direct_pointer+i) = 0x00;
447       /* Keeping the count */
448       aux = i;
449     }
450     /* Carregando ponteiros indiretos por meio do endereÃ§o de dupla
            endireiÃ§Ã£o. */
451     if(fi->d_indirect_pointer != 0x00){
452       if(lseek64(*fd, (off_t)(bsize * (fi->d_indirect_pointer)), SEEK_SET) ==
              -1){
453         fprintf(stderr, "\r%s: failure on seeking inode indirect
                block_pointers: %s\n", ARGVZERO, strerror(errno));
454         exit(-1);
455       }
456       if(verbose){
457         printf(".DI.");
458       }
459       pull_indirect_pointer = (unsigned int *) malloc(sizeof(unsigned int) *
              32);
460       frs_byte_read(fd,(pull_indirect_pointer),4);
461       i = 1;
462       while((i < 32)&&(*(pull_indirect_pointer+i-1) != 0x00)){
463         frs_byte_read(fd,(pull_indirect_pointer+i),4);
464         i++;
465       }
466       *(pull_indirect_pointer+i-1) = 0x00;
467       if(verbose){
468         printf("%d.",i);
469       }
470       /* Recebe posiÃ§Ã£o do ultimo endereco carregado no pull_direct_pointer
              .*/
471       k = aux;
472       for(j=0;(*(pull_indirect_pointer+j) != 0x00);j++){
473         if(lseek64(*fd, (off_t)(bsize * (*(pull_indirect_pointer+j))),
                SEEK_SET) == -1){
474           fprintf(stderr, "\r%s: failure on seeking inode indirect
                  block_pointers: %s\n", ARGVZERO, strerror(errno));
475           exit(-1);
476         }
477         /* Carregando endereÃ§os dos blocos diretos por meio do endereÃ§o de
                endireÃ§Ã£o simples. */
```

```
478         frs_byte_read(fd,(pull_direct_pointer+k),4);
479         l=1;
480         while((l < nbytes/4)&&(*(pull_direct_pointer+k) != 0x00)){
481           k++;
482           frs_byte_read(fd,(pull_direct_pointer+k),4);
483           l++;
484         }
485
486       }
487       *(pull_direct_pointer+k) = 0x00;
488     }
489     /* Limpando blocos indiretos do Journal. */
490     if(verbose){
491       printf("[%d loaded.][done]\n\tCleanning journal of the device.",k);
492     }
493     i=0;
494     for(i=0;(*(pull_direct_pointer+i) != 0x00);i++){
495       if(lseek64(*fd, (off_t)(bsize * (*(pull_direct_pointer+i))), SEEK_SET)
             == -1){
496         fprintf(stderr, "\r%s: failure on seeking %d inode indirect
               block_pointer %x: %s\n", ARGVZERO,i, *(pull_direct_pointer+i),
               strerror(errno));
497         exit(-1);
498       }
499       ret = ret + write(*fd, buf, bsize);
500       if((verbose)&&(((i*100/1024)%25)==0)){
501         printf(".");
502       }
503
504     }
505     if(verbose){
506       printf("[%d:%d]blocks cleared.",*(pull_direct_pointer+i),i==0?12:i);
507     }
508     /* Sync file changes to the disk.*/
509     io_f_sync(*fd, "EXT3 Journal File");
510     free(buf);
511
512     if(verbose){
513       printf(" [done]\n");
514     }
515     return  ret;
516 }
517
518
519 /* ****************************************************************
520  * journal_wipe-journal -- passa to wipe the journal file.      *
521  * ****************************************************************/
522
523 int journal_wipejournal(char *dev, unsigned int finode, struct opt_t *opt){
524     ssize_t ret = 0;
525     int fd;
526     struct frs_superblock fsb;
527     struct frs_inode fi;
528
529     if(dev_ext3_init(&fd, dev) != 0){
530       fprintf(stderr, "\r%s: cannot init rand source.\n", ARGVZERO);
```

```
531     return FAILED;
532   }
533   if(dev_ext3_track(&fd, &fsb, &fi, opt->verbose) != 0){
534     fprintf(stderr, "\r%s: cannot walk in device.\n", ARGVZERO);
535     return FAILED;
536   }
537
538   if(dev_finode_clean(&fd, fsb, finode) <= 0){
539     fprintf(stderr, "\r%s: cannot walk in device.\n", ARGVZERO);
540     return FAILED;
541   }
542
543   ret = dev_ext3_overwrite(&fd, &fsb, &fi, opt->verbose);
544
545   if(close(fd)){
546     fprintf(stderr, "\r%s: cannot close fd", ARGVZERO);
547     return FAILED;
548   }
549
550   if(ret != 0){
551     return ret;
552   }else{
553     return -1;
554   }
555 }
```

Arquivo: ./src/mcT.c

```
1  #include <time.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4
5  #include "mcT.h"
6
7  /* Global variable declaration.*/
8  static uint32_t MT[624];
9  static int index = 0;
10
11 /* Initialize the generator from a seed. */
12 void mcT_init_gen(uint32_t seed) {
13   int i;
14
15   index = 0;
16   MT[0] = seed;
17
18   for (i = 1; i<623 ; i++){
19       MT[i] = (0x6c078965 * (MT[i-1] ^ (MT[i-1] >> 30)) + i) & 0xffffffffu;
20     }
21 }
22
23 /* Generate an array of 624 untempered numbers */
24 void mcT_number_gen(){
25   int i;
26   register uint32_t y;
```

```
27    for(i=0; i<623 ;i++){
28          y = (MT[i] & 0x80000000u) + (MT[(i+1) % 624] & 0x7fffffff);
29          MT[i] = MT[(i + 397) % 624] ^ (y >> 1);
30          if ((y % 2) != 0) {
31              MT[i] = MT[i] ^ 0x9908b0df;
32          }
33      }
34 }
35
36 /* Extract a tempered pseudorandom number based on the index-th value,
37  * calling mcT_number_gen() every 624 numbers
38  */
39 uint32_t mcT_extract_number() {
40   register uint32_t y;
41
42     if (index == 0) {
43       srand(time(NULL));
44       mcT_init_gen((uint32_t)rand());
45       mcT_number_gen();
46     }
47
48     y = MT[index];
49     y ^= (y >> 11);
50     y ^= (y <<  7) & 0x9D2C5680;
51     y ^= (y << 15) & 0xEFC60000;
52     y ^= (y >> 18);
53
54     index = (index + 1) % 624;
55     return y;
56 }
```

Arquivo: ./src/frs.h

```
1 #ifndef FRS_H
2 #define FRS_H
3
4   #ifndef PATH_MAX
5     #define PATH_MAX 1023
6   #endif /* MAX */
7
8   #ifndef NAME_MAX
9     #define NAME_MAX 255
10  #endif
11
12  #ifndef ARGVZERO
13    #define ARGVZERO "frs"
14  #endif /* ARGVZERO */
15
16  #define BAD_USAGE -10
17  /* ****************** Structs ******************************** */
18
19  struct sanitizelevel {
20    /* If true enable file sanitization. */
21    int sanitize_file;
```

```
22      /* Level of file sanitization:
23       *    1 - unlink file;
24       *    2 - unlink file, clean, name, size;
25       *    3 - same of 2 and clean jornal files and inode;
26       *    4 - same of 3 and overwrites the first file block with zeros;
27       *    5 - same of 3 and overwrites the three first file blocks with
             pseudo-random values;
28       *    9 - same of 3 and overwrites the all file with pseudo-random
             values;
29       */
30      int level;
31    };
32
33
34    /* Options variable. */
35    struct opt_t {
36      /* Sanitization */
37      struct sanitizelevel slevel;
38      /* Device to clean jornal.*/
39      char *journal_dev;
40      /* Verbose*/
41      int verbose;
42      /* Change permissions to allow writing if necessary */
43      int force;
44      /* Random char source. 0 to mcTwister and 1 to /dev/random/ */
45      int random_cdev;
46    };
47
48    /* ****************** Prototypes ***************************** */
49
50    /*Catch ctr-c signal. */
51    void int_hand(int );
52
53  #endif /* FRS_H */
```

Arquivo: ./src/file.h

```
1  #include <sys/stat.h>
2
3  #include "frs.h"
4
5  enum {SECTOR_SIZE = 512};
6  enum {SECTOR_MASK = SECTOR_SIZE - 1};
7
8  #ifndef PATH_MAX
9    #define PATH_MAX 1023
10 #endif /* MAX */
11
12 /* ************************************************************
13  * filest_t - File data struct                                *
14  * ************************************************************ */
15 struct filest_t{
16   int fd;              /* file descriptor */
17   void *buf;             /* file buffer     */
```

```
  size_t bufsize;          /* buffer size      */
  char  pathname[PATH_MAX+1], /* original pathname    */
      file_name[PATH_MAX+1];  /* current filename   */
  struct stat st;          /* file status      */
};


/* ************************************************************
 * file_wipefile - Overwrite a file.                 *
 * IN:                                 *
 *    fname -    File pathname                *
 *    opt   -    Options                    *
 * OUT:     - 7: If the levels (4 or 3) and 2 and 1 works OK  *
 *        - 6: If the levels (4 or 3) and 2 works OK    *
 *        - 5: If the levels (4 or 3) and 1 works OK    *
 *        - 4: If only level (4 or 3) works OK        *
 *        - 3: If the levels 2 and 1 works OK;        *
 *        - 2: If only level 2 works OK;          *
 *        - 1: If only level 1 works OK;          *
 *        - 0: If failed                  *
 ************************************************************ */
int file_wipefile (char *fname, struct opt_t *opt);
```

Arquivo: ./src/hitface.h

```
#ifndef MAX
#define PATH_MAX 1023
#define NAME_MAX 255
#endif

#ifndef FAILED
#define FAILED -1
#endif

#ifndef O_NOFOLLOW
# define O_NOFOLLOW 0
# define SYNC 0
#endif


#ifndef ARGVZERO
#define ARGVZERO "frs"
#endif
/* ************************************************************
 * hitface_chg_name - wipe the file name.            *
 * IN:                                 *
 *    fd       - fildes of file.              *
 *    real_nema - file name.                  *
 *    nsize    - size of the file name.           *
 *    random_cdev - 0 for mercene twister and 1 to /dev/ramdom/ *
 *                                  *
 * OUT:                                  *
 *    0    - if rename OK                    *
 *    -1     - Error                       *
```

```
 * ************************************************************ */
int hitface_chg_name(int fd, char *real_name, size_t nsize, int random_cdev
    , int verbose);
```

Arquivo: ./src/dev.h

```
/* DEFINES */
#ifndef FAILED
#define FAILED -1
#endif

#ifndef ARGVZERO
#define ARGVZERO "frs"
#endif

/*const char *argvzero = "frs";*/

/* ************************************************************
 * dev_getzerobuffer - gets a buffer of size "size" and fill it   *
 * with zeros.                                    *
 *                                          *
 * IN:                                      *
 *     void *buf   -  void pointer with a pre-allocated area.*
 *     ssize_t size  -  size of buffer                *
 * OUT:                                     *
 *     void *buf   -  fill "buf" with zeros.          *
 *     ssizet n    -  number of bytes  puts in buf       *
 ************************************************************ */
ssize_t dev_getrandombuffer(void *buf, ssize_t size);


/* ************************************************************
 * dev_getzerobuffer - gets a buffer of size "size" and fill it   *
 * with zeros.                                    *
 *                                       *
 * IN:                                        *
 *     void *buf   -  void pointer with a pre-allocated area.*
 *     ssize_t size  -  size of buffer                *
 * OUT:                                     *
 *     void *buf   -  fill "buf" with zeros.          *
 *     ssizet n    -  number of bytes  puts in buf       *
 ************************************************************ */
ssize_t dev_getzerobuffer(void *buf, ssize_t size);
```

Arquivo: ./src/io.h

```
#ifndef FAILED
#define FAILED -1
#endif

#ifndef ARGVZERO
#define ARGVZERO "frs"
#endif
```

```
8
9  /* ****************************************************************
10  * io_dir_sync - Sync directory changes to the disk.          *
11  * IN:                                    *
12  *     ffd    - fildes of file                    *
13  *     name   - File pathname                     *
14  * OUT:                                   *
15  *     0    - Sync OK                       *
16  *     -1     - Error                         *
17  * ***************************************************************** */
18 int io_dir_sync(int ffd, char *name);
19
20 /* ****************************************************************
21  * io_f_sync - Sync file changes to the disk.           *
22  * IN:                                    *
23  *     fd     - fildes of file                    *
24  *     name   - File pathname                     *
25  * OUT:                                   *
26  *     0    - Sync OK                       *
27  *     -1     - Error                         *
28  * ***************************************************************** */
29 int io_f_sync(int fd, char *name);
```

Arquivo: ./src/inode.h

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  #include <stdlib.h>
6  #include <sys/stat.h>
7
8  #include <ctype.h>
9  #include <errno.h>
10 #include <assert.h>
11
12 #include "journal.h"
13
14 #ifndef FAILED
15 #define FAILED -1
16 #endif
17
18 #ifndef ARGVZERO
19 #define ARGVZERO "frs"
20 #endif
21
22 /* ****************************************************************
23  * calc_inodetb_seek -- calculate the seek to inode table.    *
24  *                                        *
25  * INPUT:                                   *
26  *     off_t    -    FS Block Size                  *
27  *     off_t - Group Number of the inode             *
28  *                                        *
29  * OUTPUT:                                    *
```

```
 *     off_t - Seek address to the inode table          *
 * ************************************************************ */
off_t calc_inodetb_seek(off_t , off_t );


/* ************************************************************
 * calc_inode_seek -- calculate the seek to inode.           *
 *                                  *
 * INPUT:                             *
 *     off_t   -   FS Block Size                  *
 *     off_t - Address to the begin of inode table       *
 *     off_t   -   Inode size                     *
 *     off_t - Position of the inode in the group        *
 * OUTPUT:                                *
 *     off_t - calculate the seek to inode.          *
 * ************************************************************ */
off_t calc_inode_seek(off_t , off_t , off_t , off_t );

/* ************************************************************
 * dev_finode_clean -- clear the inode of a device           *
 *                                  *
 * INPUT:                               *
 *     int *fd           - device filde;       *
 *     struct frs_superblock fsb - superblock struct data;   *
 *     int finode          - inode number;       *
 *                                  *
 * OUTPUT:                                  *
 *     2*inode size        - if ok.              *
 *     -1              - if got problems.        *
 *     exit            - if couldn't seek        *
 * ************************************************************ */
int dev_finode_clean(int *, struct frs_superblock , int );
```

Arquivo: ./src/journal.h

```
#include <stdio.h>
#include <string.h>   /* See feature_test_macros(7) */
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>

#include <ctype.h>
#include <errno.h>
#include <assert.h>

#include "frs.h"


#ifndef FAILED
#define FAILED -1
#endif

```

```
20 #ifndef ARGVZERO
21 #define ARGVZERO "frs"
22 #endif
23
24 #ifndef JOURNAL_STRUCTS
25   /* ************************************************************
26    * Super block structure                                      *
27    * ************************************************************/
28   struct frs_superblock {
29     /* Number of inodes in file system: bytes 0-3 */
30     unsigned int total_inodes_number;
31     /* Number of blocks in file system: bytes 4-7 */
32     unsigned int total_blocks_number;
33     /* Number of blocks reserved to prevent fs filling up: bytes 8-11 */
34     unsigned int total_blocks_reserved;
35     /* Block where block group 0 starts: bytes 20-23 */
36     unsigned int start_block_group_zero;
37     /* Block size bytes 24-27 */
38     unsigned int fs_block_size;
39     /* Fragment size bytes 28-31 */
40     unsigned int fs_fragment_size;
41     /* Number of blocks per group bytes 32-35 */
42     unsigned int blocks_per_group;
43     /* Number of fragments per group bytes 36-39 */
44     unsigned int fragments_per_group;
45     /* Number of inodes per group bytes 40-43 */
46     unsigned int inodes_per_group;
47     /* Signature bytes 56-57 */
48     unsigned int fs_signature;
49     /* OS creator bytes 72-75 */
50     unsigned int fs_os_creator;
51     /* Size of each inode structure 88-89 */
52     unsigned int fs_inode_size;
53     /* Algorithm usage bitmap: bytes 200-203 */
54     unsigned int bitmap_algoritm;
55     /* Journal inode: bytes 224-227 */
56     unsigned int fs_journal_inode;
57     /* Journal device: bytes 228-231 */
58     unsigned int fs_journal_device;
59   };
60
61
62   /* ************************************************************
63    * Inode structure                                            *
64    * ************************************************************/
65   struct frs_inode{
66     /* Access Time: bytes 8-11 */
67     unsigned int a_time;
68     /* Change Time: bytes 12-15 */
69     unsigned int c_time;
70     /* Modification time: bytes 16-19 */
71     unsigned int m_time;
72     /* Deletion time: bytes 20-23 */
73     unsigned int d_time;
74     /* 12 direct block pointers: bytes 40-87 */
75     unsigned int block_pointer1;
```

```
76      unsigned int block_pointer2;
77      unsigned int block_pointer3;
78      unsigned int block_pointer4;
79      unsigned int block_pointer5;
80      unsigned int block_pointer6;
81      unsigned int block_pointer7;
82      unsigned int block_pointer8;
83      unsigned int block_pointer9;
84      unsigned int block_pointer10;
85      unsigned int block_pointer11;
86      unsigned int block_pointer12;
87      /* 1 single indirect block pointer: bytes 88-91 */
88      unsigned int s_indirect_pointer;
89      /* 1 double indirect block pointer: bytes 92-95  */
90      unsigned int d_indirect_pointer;
91      /* 1 triple indirect block pointer: bytes 96-99 */
92      unsigned int t_indirect_pointer;
93    };
94
95    struct journal_sb{
96      /* Journal signature: bytes 0-3 */
97      unsigned int signature;
98      /* Block type (1 - Descriptor,2 Commit,3 SB v1,4
99       * SB v2,5 Revoke): bytes 4-7 */
100     unsigned int block_type;
101     /* Sequencial number: bytes 8-11 */
102     unsigned int seq_num;
103     /* Journal block size: bytes 12-15 */
104     unsigned int block_size;
105     /* Number of journal blocks: bytes 16-19 */
106     unsigned int qtd_blocks;
107     /* Journal block where the journal actually starts: bytes 20-23 */
108     unsigned int jstart_block;
109     /* Sequence number of first transaction: bytes 24-27 */
110     unsigned int seq_tr;
111     /* Journal block of first transaction: bytes 28-31 */
112     unsigned int bfst_tr;
113     /* Error codes */
114     unsigned int erron_code;
115     /* Compatible fealures */
116     unsigned int cfealure;
117     /* Imcompatible fealures */
118     unsigned int ifealure;
119     /* Read only fealures */
120     unsigned int rofealure;
121     /* Journal UUID. */
122     unsigned int uuid;
123     /* Number of file system using the journal. */
124     unsigned int fsqtd;
125     /* Location of super block copy. */
126     unsigned int sbcopy;
127     /* Max number of journal blocks per transaction. */
128     unsigned int max_jbpt;
129     /* Max number of file system blocks per transaction. */
130     unsigned int max_fsbpt;
131   };
```

```
132   #define JOURNAL_STRUCTS 1
133 #endif
134 /* ****************************************************************
135  * journal_wipe-journal - Level 3 clean up. Clean journal file. *
136  *                                      *
137  * IN:                                    *
138  *     char*       -  device.                      *
139  *     unsigned int -  inode number of a file              *
140  * OUT:                                   *
141  *     0    - If ok                       *
142  *     -1     - If FAILED                      *
143  * ***************************************************************** */
144 int journal_wipejournal(char*, unsigned int, struct opt_t *);
145
146 int frs_byte_read(int *fd, unsigned int *buf, size_t nsize);
```

Arquivo: ./src/mcT.h

```
1
2 uint32_t mcT_extract_number();
```

Referências

[1] B. Carrie. *File System Forensic Analysis.* Addison Wesley Professional, 3 2005. vii, ix, 4, 5, 7, 8, 21, 22, 23, 25, 26, 27, 34, 37, 38, 43

[2] F. Cohen. A note on recovery of data from overwritten areas of magnetic media. http://all.net/ForensicsPapers/2012-12-07-OverwrittenMagneticRecovery.pdf, 12 2012. última vizualização em: 24.07.2014. 66

[3] Wikipedia: The Free Encyclopedia. Flash memory. http://en.wikipedia.org/wiki/Flash_memory, 6 2013. última visualização em: 07.06.2013. 5

[4] Wikipedia: The Free Encyclopedia. Mersenne twister. http://en.wikipedia.org/wiki/Mersenne_twister, 6 2014. última visualização em: 07.06.2014.

[5] D. Farmer and W. Venema. Forensic computer analysis: An introduction, 9 2000. última visualização em: 13.06.2013. vii, 20, 32

[6] P. Festa and L. M. Bowman. Computers hinder paper shredders. http://news.cnet.com/2100-1023-829004.html, 12 2012. última visualização em: 02.06.2013. 1

[7] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Sixth USENIX Security Symposium Proceedings*, pages 77–90, July 1996. 1, 3

[8] N. Halverson. Digital memory uses 100 times less power. http://news.discovery.com/tech/alternative-power-sources/digital-memory-uses-100-times-less-power-110316.htm, 12 2000. última visualização em: 17.07.2013. 6

[9] A. Merola. *Data Carving Concepts.* SANS Institute, 11 2008. 40

[10] G. Narváez. *Taking Advantage of EXT3 journaling file system in forensic investigation.* SANS Institute, 12 2007. 53, 55, 57

[11] NTFS.com. Ntfs basics. http://www.ntfs.com/ntfs_basics.htm, 9 2001. última visualização em: 08.06.2013. vii, 14, 15

[12] Oxford University Press. Oxford dictionaries, 06 2013. última visualização em: 13.06.2013. 12

[13] T. L. I. Project. Inode definition, 6 2005. última visualização em: 24.04.2013. 16

[14] Andrea Röck. Pseudorandom number generators for cryptographic applications. Master's thesis, Paris-Lodron-Universität Salzburg, 3 2005. 44

[15] M. E. Russinovich and D. A. Solomon. *Windows Internals*. Microsoft Press, Redmond, WA, fifth edition, 2009. 16

[16] National Information Systems Security. Sensitive information. http://www.its.bldrdoc.gov/fs-1037/dir-032/_4768.htm, 8 1996. última visualização em: 02.06.2013. 1

[17] A. Sheppard. How to read a smashed hard drive. http://www.popularmechanics.com/technology/how-to/computer-security/how-to-read-a-smashed-hard-drive-14877558, 12 2012. última visualização em: 02.06.2013. 1

[18] W. Stallings. *Arquitetura e Organização de Computadores*. Prentice Hall, São Paulo, quinta edition, 2003. 3

[19] A. S. Tanenbaum and A. S. Woodhull. *Sistemas Operacionais: Projeto e Implementação*. Bookman, Porto Alegre, terceira edition, 2008. vii, 8, 13, 14

[20] W. Venema. File recovery techniques, 12 2000. última visualização em: 13.06.2013. 19

[21] C. Wright. Overwriting hard drive data: The great wiping controversy. http://digital-forensics.sans.org/blog/2009/01/15/overwriting-hard-drive-data/, 01 2009. última vizualização em: 24.07.2014. 67