

UNIVERSIDADE XYZ

FACULDADE DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Análise e Implementação de uma Ferramenta para Formato ELF

AUTOR DO TRABALHO

Orientador: Prof. Dr. Nome do Orientador

Local

2025

Resumo

Este trabalho apresenta um estudo detalhado sobre o formato de arquivo ELF (Executable and Linkable Format), amplamente utilizado em sistemas operacionais Unix-like como formato padrão para executáveis, bibliotecas compartilhadas, arquivos objeto e core dumps. A pesquisa abrange desde conceitos fundamentais de arquitetura de computadores e sistemas operacionais até uma análise minuciosa da estrutura interna dos arquivos ELF, incluindo cabeçalhos, segmentos e seções.

O trabalho também descreve a implementação de uma ferramenta para análise e manipulação de arquivos ELF, comparando-a com soluções existentes como elfutils. São discutidas aplicações práticas dessa ferramenta, como detecção de software malicioso e patching de binários.

Os resultados obtidos demonstram a importância do entendimento aprofundado do formato ELF para áreas como desenvolvimento de sistemas, segurança computacional e engenharia reversa de software. A ferramenta desenvolvida oferece uma alternativa moderna e flexível para trabalhar com arquivos ELF em ambientes de desenvolvimento e pesquisa.

Palavras-chave: ELF, sistemas operacionais, formato de arquivo, análise binária, compiladores, linkers.

Abstract

Este trabalho apresenta um estudo detalhado do formato ELF (Executable and Linkable Format), amplamente utilizado em sistemas operacionais Unix-like como o formato padrão para executáveis, bibliotecas compartilhadas, arquivos objeto e core dumps. A pesquisa abrange desde conceitos fundamentais de arquitetura de computadores e sistemas operacionais até uma análise minuciosa da estrutura interna dos arquivos ELF, incluindo cabeçalhos, segmentos e seções.

O trabalho também descreve a implementação de uma ferramenta para análise e manipulação de arquivos ELF, comparando-a com soluções existentes como elfutils. São discutidas aplicações práticas dessa ferramenta, como detecção de software malicioso e patching de binários.

Os resultados demonstram a importância do entendimento aprofundado do formato ELF para áreas como desenvolvimento de sistemas, segurança computacional e engenharia reversa de software. A ferramenta desenvolvida oferece uma alternativa moderna e flexível para trabalhar com arquivos ELF em ambientes de desenvolvimento e pesquisa.

Keywords: ELF, operating systems, file format, binary analysis, compilers, linkers.

Contents

1	Introdução	8
1.1	Conceitos Fundamentais de Arquitetura de Computadores	9
1.1.1	Arquitetura de von Neumann	9
1.1.2	Representação Binária	9
1.1.3	Conjunto de Instruções (ISA)	10
1.1.4	Ciclo de Execução de Instruções	11
1.1.5	Registradores e Memória	11
1.2	Conceitos de Memória e Armazenamento	12
1.2.1	Hierarquia de Memória	12
1.2.2	Memória Virtual	13
1.2.3	Segmentação de Memória	14
1.2.4	Alinhamento de Memória	14
1.2.5	Endianess	15
2	Formato ELF	16
2.1	Padrão TIS (Tool Interface Standard) para ELF	17
2.1.1	Histórico e Desenvolvimento	17
2.1.2	Objetivos do Padrão TIS	17
2.1.3	Alcance da Especificação	17
2.1.4	Implementação em Diferentes Sistemas	18
2.1.5	Documentação TIS	18
2.1.6	Evolução e Versões	18
2.2	Cabeçalho ELF (ELF Header)	20
2.2.1	Estrutura do Cabeçalho ELF	20
2.2.2	Campo de Identificação (e_ident)	20
2.2.3	Tipo de Arquivo (e_type)	21
2.2.4	Arquitetura Alvo (e_machine)	21
2.2.5	Pontos de Entrada e Tabelas (e_entry, e_phoff, e_shoff)	22
2.2.6	Contagens e Tamanhos (e_phnum, e_shnum, etc.)	22
2.2.7	Implementação Prática: Análise de um Cabeçalho ELF	23

2.2.8	Considerações de Segurança	23
2.2.9	Relação com Outras Partes do ELF	24
2.3	Segmentos ELF (Program Headers)	25
2.3.1	Função dos Segmentos	25
2.3.2	Estrutura do Program Header	25
2.3.3	Tipos de Segmentos (p_type)	26
2.3.4	Flags de Segmento (p_flags)	26
2.3.5	Mapeamento de Arquivo para Memória	27
2.3.6	Segmentos PT_LOAD	27
2.3.7	Segmento PT_INTERP	28
2.3.8	Segmento PT_DYNAMIC	28
2.3.9	Alinhamento de Segmentos (p_align)	28
2.3.10	Análise de Segmentos com Ferramentas	28
2.3.11	Relação entre Segmentos e Seções	29
2.3.12	Considerações para Diferentes Arquiteturas	30
2.4	Seções ELF (Section Headers)	31
2.4.1	Conceito e Propósito	31
2.4.2	Estrutura do Section Header	31
2.4.3	Tipos de Seção	32
2.4.4	Flags de Seção	32
2.4.5	Seções Especiais	32
2.4.6	Tabela de Strings da Seção	33
2.4.7	Relação com Segmentos	33
2.4.8	Análise de Seções com Ferramentas	34
2.4.9	Seções de Depuração	34
2.4.10	Uso de Seções em Análise Binária	35
2.4.11	Extensões Específicas	35
3	Análise de Código	36
4	Análise Comparativa	37
5	Aplicações Futuras	38
6	Conclusão	39
7	Referências	40

List of Figures

1.1	Diagrama simplificado da arquitetura de von Neumann	9
1.2	Hierarquia de memória em sistemas computacionais modernos	12
2.1	Estrutura do cabeçalho ELF e sua relação com o resto do arquivo	22
2.2	Mapeamento de segmentos ELF do arquivo para a memória	26
2.3	Organização das seções em um arquivo ELF típico	33

List of Tables

1.1	Exemplos de representação binária para diferentes tipos de dados	10
1.2	Comparação entre sistemas com e sem memória virtual	13
2.1	Campos do array e_ident	21
2.2	Segmentos típicos em um arquivo ELF executável	27
2.3	Seções comuns em arquivos ELF	33

Chapter 1

Introdução

1.1 Conceitos Fundamentais de Arquitetura de Computadores

A compreensão do formato ELF (Executable and Linkable Format) depende fundamentalmente do entendimento da arquitetura de computadores e de como os programas são representados e executados em um sistema computacional. Esta seção apresenta os conceitos básicos necessários para esse entendimento.

1.1.1 Arquitetura de von Neumann

A maioria dos computadores modernos é baseada na arquitetura de von Neumann, proposta por John von Neumann em 1945. Essa arquitetura define um computador com:

- **Unidade Central de Processamento (CPU):** responsável por buscar instruções da memória, decodificá-las e executá-las.
- **Unidade de Memória:** armazena tanto dados quanto instruções.
- **Dispositivos de Entrada e Saída:** permitem a comunicação do computador com o mundo externo.
- **Barramento:** conecta os componentes, permitindo o fluxo de dados entre eles.

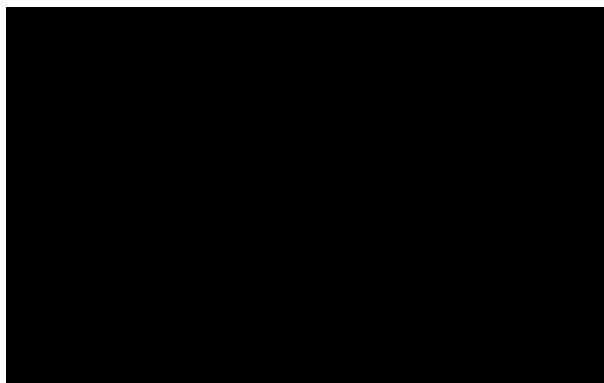


Figure 1.1: Diagrama simplificado da arquitetura de von Neumann

Como podemos observar na Figura 1.1, a arquitetura de von Neumann estabelece uma conexão direta entre a CPU e a memória, formando a base para o funcionamento dos computadores modernos. Esta arquitetura será importante para entendermos como os programas no formato ELF são carregados e executados, como veremos em Capítulo 2.

1.1.2 Representação Binária

Na essência de qualquer computador digital está a representação binária de informações. Todos os dados e instruções são codificados como sequências de bits (0s e 1s). Um byte,

composto de 8 bits, é a unidade básica de armazenamento na maioria dos computadores modernos.

Múltiplas representações podem ser derivadas dessa codificação binária:

- **Valores inteiros:** representados diretamente em binário ou usando complemento de dois para números negativos.
- **Caracteres:** codificados usando padrões como ASCII ou UTF-8.
- **Instruções de máquina:** sequências específicas de bits que a CPU interpreta como comandos.

Table 1.1: Exemplos de representação binária para diferentes tipos de dados

Tipo de Dado	Valor	Representação Binária
Inteiro positivo	42	00101010
Inteiro negativo	-42	11010110 (complemento de dois)
Caractere ASCII	'A'	01000001
Instrução x86	ADD AL, BL	00000000

A Tabela 1.1 ilustra como diferentes tipos de dados são representados em formato binário. Esta compreensão da representação binária é fundamental para a análise de arquivos ELF, que será discutida em detalhes na Seção 2.2 do Capítulo 2.

1.1.3 Conjunto de Instruções (ISA)

O ISA (Instruction Set Architecture) define as instruções que o processador pode executar, incluindo operações aritméticas, lógicas, de transferência de dados e de controle de fluxo. O ISA representa a interface entre software e hardware, determinando como os programas são escritos e compilados para serem executados na máquina.

Existem dois paradigmas principais de ISA:

- **CISC (Complex Instruction Set Computer):** oferece instruções complexas e poderosas, como o x86.
- **RISC (Reduced Instruction Set Computer):** utiliza instruções mais simples e uniformes, como ARM e RISC-V.

Como veremos na Seção ??, a arquitetura x86-64 é particularmente relevante para nossa discussão sobre o formato ELF em sistemas Linux modernos.

1.1.4 Ciclo de Execução de Instruções

O ciclo básico de execução em um processador consiste em:

1. **Fetch**: buscar a próxima instrução da memória
2. **Decode**: identificar a operação a ser realizada
3. **Execute**: realizar a operação
4. **Write-back**: armazenar os resultados, se necessário

Este ciclo, conhecido como ciclo de busca-execução, é fundamental para entender como programas são executados e, conseqüentemente, como o formato ELF é estruturado para permitir essa execução. A relação entre este ciclo e a organização de memória discutida na Seção 1.2 será essencial para compreender como o sistema operacional carrega binários ELF.

1.1.5 Registradores e Memória

Processadores possuem registradores, pequenas unidades de memória de acesso rápido dentro da CPU:

- **Registradores de Uso Geral**: armazenam dados temporários durante processamento
- **Registrador de Instrução**: mantém a instrução atual sendo executada
- **Contador de Programa (PC)**: aponta para a próxima instrução a ser executada
- **Ponteiro de Pilha (SP)**: gerencia a pilha de execução

Compreender a relação entre registradores e memória é essencial para entender como os programas compilados no formato ELF são carregados e executados pelo sistema operacional. Esta relação será explorada em maior detalhe no Capítulo 3 quando discutirmos a análise de código assembly em arquivos ELF.

1.2 Conceitos de Memória e Armazenamento

Em um sistema computacional, a memória é um componente crítico que afeta diretamente o desempenho e as capacidades do sistema. Compreender seus diferentes tipos e hierarquias é fundamental para entender como os programas em formato ELF são carregados e executados.

1.2.1 Hierarquia de Memória

Os sistemas computacionais modernos implementam uma hierarquia de memória para equilibrar velocidade, capacidade e custo:

- **Registradores:** Extremamente rápidos, localizados no processador, com capacidade muito limitada.
- **Cache:** Memória intermediária de alta velocidade que armazena cópias de dados frequentemente acessados.
 - **Cache L1:** Menor e mais rápido, geralmente dividido em cache de instruções e cache de dados.
 - **Cache L2:** Maior e um pouco mais lento que o L1, geralmente unificado.
 - **Cache L3:** Presente em processadores mais modernos, compartilhado entre núcleos.
- **Memória Principal (RAM):** Armazena programas e dados ativamente em uso, com acesso mais lento que caches.
- **Armazenamento Secundário:** Dispositivos não voláteis como SSDs e HDDs, com grande capacidade mas acesso significativamente mais lento.

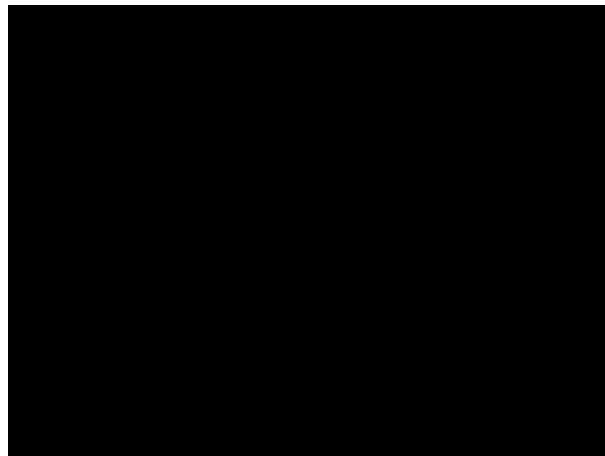


Figure 1.2: Hierarquia de memória em sistemas computacionais modernos

Como mostrado na Figura 1.2, esta hierarquia é essencial para o desempenho do sistema. Os conceitos apresentados na Seção 1.1 combinados com esta hierarquia de memória influenciam diretamente o desenho do formato ELF.

1.2.2 Memória Virtual

Conceito e Função

A memória virtual é uma abstração que separa o espaço de endereçamento lógico (visto pelos processos) do espaço de endereçamento físico (hardware real). Suas principais funções incluem:

- Permitir que programas operem como se tivessem mais memória disponível que a RAM física.
- Proteger processos, impedindo que acessem áreas de memória de outros processos.
- Facilitar o compartilhamento de memória e bibliotecas entre processos diferentes.

Paginação

A memória virtual geralmente é implementada usando paginação:

- O espaço de endereçamento é dividido em unidades de tamanho fixo chamadas páginas.
- O mapeamento entre páginas virtuais e quadros físicos é mantido em tabelas de páginas.
- Páginas não utilizadas podem ser temporariamente transferidas para armazenamento secundário (swap).
- O hardware MMU (Memory Management Unit) traduz endereços virtuais em físicos durante a execução.

Table 1.2: Comparação entre sistemas com e sem memória virtual

Característica	Com Memória Virtual	Sem Memória Virtual
Limitação de memória	Endereçamento virtual	RAM física disponível
Proteção de memória	Sim, por processo	Limitada ou inexistente
Compartilhamento	Eficiente via mapeamento	Complicado
Fragmentação	Gerenciada via paginação	Problemática

Veremos em Seção 2.3 como o formato ELF é projetado para funcionar eficientemente com sistemas de memória virtual, especificando como segmentos devem ser carregados na memória.

1.2.3 Segmentação de Memória

A segmentação é um esquema de gerenciamento de memória que divide o espaço de endereçamento em segmentos lógicos:

- **Segmento de Código:** Contém as instruções executáveis do programa (somente leitura).
- **Segmento de Dados:** Armazena variáveis globais e estáticas inicializadas.
- **Segmento BSS (Block Started by Symbol):** Contém variáveis globais e estáticas não inicializadas.
- **Heap:** Área para alocação dinâmica de memória durante a execução.
- **Stack:** Armazena variáveis locais, parâmetros de função e informações de controle.

Este conceito de segmentação é particularmente relevante para o formato ELF, que organiza seções de código e dados em segmentos para carregamento eficiente. Esta organização será explorada em detalhes no Capítulo 2.

1.2.4 Alinhamento de Memória

O alinhamento de memória refere-se à forma como os dados são organizados em endereços que são múltiplos de seu tamanho:

- Instrução ou dado de 4 bytes geralmente é alinhado em endereços múltiplos de 4.
- Dados de 8 bytes geralmente são alinhados em endereços múltiplos de 8.

O alinhamento adequado é crucial para:

- Melhorar a eficiência de acesso à memória.
- Garantir compatibilidade com arquiteturas que exigem alinhamento.
- Otimizar o desempenho do sistema.

No formato ELF, o alinhamento é especificado para diferentes seções e segmentos, garantindo que quando carregados na memória, eles mantenham o alinhamento necessário para execução eficiente. Como será detalhado na Seção 2.4, esta é uma propriedade fundamental do formato.

1.2.5 Endianess

Endianess refere-se à ordem em que bytes de dados multi-byte são armazenados na memória:

- **Little-endian:** O byte menos significativo é armazenado no endereço mais baixo.
- **Big-endian:** O byte mais significativo é armazenado no endereço mais baixo.

Arquiteturas diferentes usam convenções diferentes (Intel x86 usa little-endian, enquanto algumas versões do PowerPC usam big-endian). O formato ELF inclui informações sobre endianess no seu cabeçalho, permitindo que sistemas interpretem corretamente os dados binários independentemente da arquitetura, como será discutido na Seção [2.2](#) no próximo capítulo.

Chapter 2

Formato ELF

2.1 Padrão TIS (Tool Interface Standard) para ELF

O formato ELF (Executable and Linkable Format) é definido pelo padrão TIS (Tool Interface Standard), estabelecido pelo Tool Interface Standards Committee. Este padrão define detalhadamente a estrutura e organização dos arquivos binários no formato ELF, permitindo que ferramentas como compiladores, linkers, carregadores e depuradores possam trabalhar de maneira coerente com esses arquivos.

2.1.1 Histórico e Desenvolvimento

O formato ELF foi originalmente desenvolvido pela USL (UNIX System Laboratories) como parte do ABI (Application Binary Interface) do System V Release 4 (SVR4). Em 1993, o comitê TIS (Tool Interface Standard) publicou a especificação ELF oficial, que foi amplamente adotada pelas distribuições Unix e Unix-like, incluindo Linux, BSD, Solaris, HP-UX, entre outros.

Antes do ELF, sistemas Unix utilizavam formatos como a.out e COFF (Common Object File Format), mas o ELF foi projetado para superar as limitações desses formatos anteriores e prover maior flexibilidade e extensibilidade.

2.1.2 Objetivos do Padrão TIS

A especificação TIS para ELF foi desenvolvida com os seguintes objetivos:

- Estabelecer um formato de arquivo objeto que pudesse ser usado em diferentes arquiteturas de processadores.
- Permitir operações eficientes de linking estático e dinâmico.
- Padronizar a interface entre compiladores, assemblers, linkers e sistemas operacionais.
- Facilitar a análise e manipulação de binários por ferramentas de desenvolvimento.
- Suportar extensão para novas arquiteturas e funcionalidades.

2.1.3 Alcance da Especificação

A especificação TIS para ELF define:

- Estrutura do cabeçalho ELF (ELF Header)
- Formato das tabelas de programa (Program Headers)
- Formato das tabelas de seção (Section Headers)
- Tipos de seções e seus conteúdos

- Tabelas de símbolos e relocações
- Convenções para informações de depuração
- Extensões específicas de arquitetura

2.1.4 Implementação em Diferentes Sistemas

Embora o ELF seja um padrão, diferentes sistemas operacionais e arquiteturas implementam extensões específicas:

- **Linux:** Implementa extensões específicas para suas funcionalidades, como suporte a segurança SELinux e capacidades.
- **FreeBSD/NetBSD:** Adicionam extensões para suas características específicas.
- **Solaris:** Implementa extensões adicionais para suportar funcionalidades específicas da plataforma.
- **ARM/MIPS/PowerPC:** Possuem especificações complementares para lidar com características específicas dessas arquiteturas.

2.1.5 Documentação TIS

A especificação TIS para ELF é documentada em vários documentos, incluindo:

- Especificação genérica ELF (TIS ELF 1.2)
- Especificações específicas de processador (como IA-32, AMD64, ARM, etc.)
- ABIs específicos de sistema operacional

O documento base, conhecido como "Tool Interface Standard (TIS) Executable and Linkable Format (ELF) Specification", define a estrutura fundamental do formato, enquanto documentos adicionais cobrem extensões e adaptações específicas para diferentes arquiteturas e sistemas.

2.1.6 Evolução e Versões

A especificação ELF evoluiu ao longo do tempo para acomodar novas funcionalidades e arquiteturas:

- ELF 1.0: Versão inicial para SVR4
- ELF 1.1: Adicionou suporte para executáveis dinâmicos

- ELF 1.2: Incluiu extensões e clarificações importantes
- Versões posteriores: Extensões para 64 bits, novos processadores e funcionalidades específicas de sistema

Em sistemas modernos, a especificação é mantida por várias entidades, incluindo a Linux Foundation, grupos de desenvolvimento BSD, e empresas como Oracle (para Solaris) e ARM.

Esta padronização, embora com variações específicas para diferentes sistemas, permite a portabilidade de binários entre sistemas compatíveis e facilita o desenvolvimento de ferramentas que manipulam arquivos binários em diversos ambientes.

2.2 Cabeçalho ELF (ELF Header)

O cabeçalho ELF é a estrutura inicial e mais fundamental de um arquivo ELF, fornecendo informações essenciais sobre o formato do arquivo, sua arquitetura alvo e pontos de entrada para processamento adicional. Este cabeçalho serve como um mapa que permite ao sistema operacional e outras ferramentas interpretarem corretamente o conteúdo do arquivo.

2.2.1 Estrutura do Cabeçalho ELF

O cabeçalho ELF é uma estrutura de dados que ocupa os primeiros bytes do arquivo. Na especificação TIS, conforme mencionado na Seção ??, o cabeçalho é definido por uma estrutura chamada `Elf32_Ehdr` (para arquivos ELF de 32 bits) ou `Elf64_Ehdr` (para arquivos ELF de 64 bits).

```
1 typedef struct {
2     unsigned char e_ident[EI_NIDENT]; /* Identificacao do arquivo ELF */
3     Elf64_Half    e_type;              /* Tipo de objeto */
4     Elf64_Half    e_machine;          /* Arquitetura necessaria */
5     Elf64_Word    e_version;          /* Versao do objeto */
6     Elf64_Addr    e_entry;            /* Endereco de entrada virtual */
7     Elf64_Off     e_phoff;            /* Offset da tabela de program header
8     */
9     Elf64_Off     e_shoff;            /* Offset da tabela de section header
10    */
11    Elf64_Word     e_flags;            /* Flags especificas do processador
12    */
13    Elf64_Half     e_ehsize;           /* Tamanho do ELF header */
14    Elf64_Half     e_phentsize;        /* Tamanho de uma entrada da program
15    header table */
16    Elf64_Half     e_phnum;            /* Numero de entradas na program
17    header table */
18    Elf64_Half     e_shentsize;        /* Tamanho de uma entrada da section
19    header table */
20    Elf64_Half     e_shnum;            /* Numero de entradas na section
21    header table */
22    Elf64_Half     e_shstrndx;         /* Indice da tabela de secoes que
23    contem nomes de secoes */
24 } Elf64_Ehdr;
```

Listing 2.1: Estrutura do cabeçalho ELF de 64 bits

2.2.2 Campo de Identificação (e_ident)

Os primeiros bytes do cabeçalho ELF, o array `e_ident`, são particularmente importantes e contêm informações para identificação do arquivo:

Table 2.1: Campos do array `e_ident`

Índice	Nome	Descrição
0-3	EI_MAG0 até EI_MAG3	"Magic number": 0x7F, 'E', 'L', 'F'
4	EI_CLASS	Classe de arquivo (32 ou 64 bits)
5	EI_DATA	Endianness dos dados (little/big-endian)
6	EI_VERSION	Versão do ELF (geralmente 1)
7	EI_OSABI	ABI do sistema operacional alvo
8	EI_ABIVERSION	Versão da ABI específica
9-15	EI_PAD	Bytes reservados, preenchidos com zeros

O "Magic Number" no início do arquivo (0x7F, 'E', 'L', 'F') é especialmente importante, pois permite que sistemas identifiquem rapidamente se um arquivo é do formato ELF ou não.

2.2.3 Tipo de Arquivo (`e_type`)

O campo `e_type` especifica o tipo de arquivo ELF. Os valores mais comuns são:

- `ET_NONE` (0): Tipo não especificado
- `ET_REL` (1): Arquivo relocável (objeto)
- `ET_EXEC` (2): Arquivo executável
- `ET_DYN` (3): Objeto compartilhado (biblioteca compartilhada)
- `ET_CORE` (4): Arquivo core (dump de memória)

Esta classificação afeta diretamente como o arquivo será tratado pelo sistema operacional, como veremos em Capítulo 3 quando discutirmos o carregamento de arquivos ELF.

2.2.4 Arquitetura Alvo (`e_machine`)

O campo `e_machine` especifica a arquitetura para a qual o arquivo foi compilado. Alguns valores comuns incluem:

- `EM_386` (3): Intel 80386
- `EM_X86_64` (62): AMD x86-64
- `EM_ARM` (40): ARM
- `EM_AARCH64` (183): ARM 64-bits (AArch64)
- `EM_RISCV` (243): RISC-V

Este campo é crítico para o sistema operacional determinar se o binário é compatível com o hardware em que está sendo executado. Como discutido na Seção 1.1.3, o ISA do processador deve corresponder àquele para o qual o binário foi compilado.

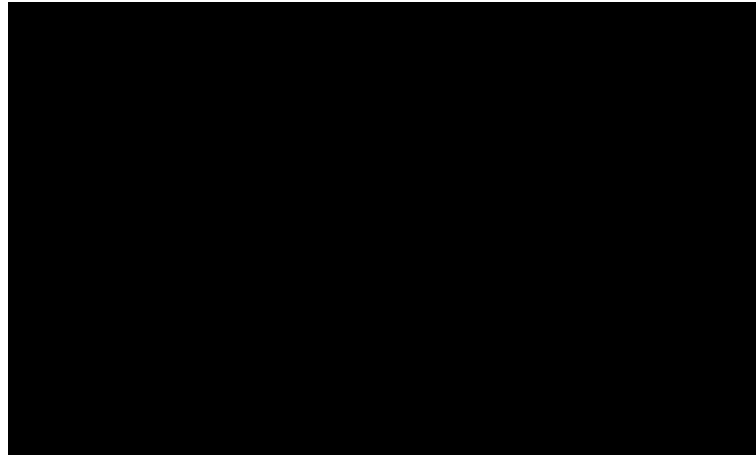


Figure 2.1: Estrutura do cabeçalho ELF e sua relação com o resto do arquivo

2.2.5 Pontos de Entrada e Tabelas (`e_entry`, `e_phoff`, `e_shoff`)

Os campos `e_entry`, `e_phoff`, e `e_shoff` são essenciais para o carregamento e execução do arquivo:

- `e_entry`: Endereço virtual do ponto de entrada do programa, onde a execução deve começar.
- `e_phoff`: Offset, em bytes, do início da tabela de cabeçalhos de programa (Program Header Table).
- `e_shoff`: Offset, em bytes, do início da tabela de cabeçalhos de seção (Section Header Table).

Como veremos nas seções Seção 2.3 e Seção 2.4, estas tabelas fornecem informações detalhadas sobre como o arquivo deve ser carregado na memória e como suas partes estão organizadas.

2.2.6 Contagens e Tamanhos (`e_phnum`, `e_shnum`, etc.)

Os campos restantes fornecem informações sobre o número e tamanho das entradas nas tabelas de cabeçalho do programa e de seção:

- `e_ehsize`: Tamanho do cabeçalho ELF em bytes.
- `e_phentsize`: Tamanho de cada entrada na Program Header Table.

- `e_phnum`: Número de entradas na Program Header Table.
- `e_shentsize`: Tamanho de cada entrada na Section Header Table.
- `e_shnum`: Número de entradas na Section Header Table.
- `e_shstrndx`: Índice da seção na Section Header Table que contém os nomes das seções.

Estes campos permitem que o sistema operacional ou outras ferramentas naveguem corretamente pelo arquivo e localizem informações específicas.

2.2.7 Implementação Prática: Análise de um Cabeçalho ELF

Na prática, podemos analisar o cabeçalho ELF usando ferramentas como `readelf`:

```

1 $ readelf -h /bin/ls
2 ELF Header:
3   Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
4   Class:                               ELF64
5   Data:                               2's complement, little endian
6   Version:                             1 (current)
7   OS/ABI:                               UNIX - System V
8   ABI Version:                          0
9   Type:                                  DYN (Shared object file)
10  Machine:                               Advanced Micro Devices X86-64
11  Version:                               0x1
12  Entry point address:                   0x6050
13  Start of program headers:              64 (bytes into file)
14  Start of section headers:             137240 (bytes into file)
15  Flags:                                  0x0
16  Size of this header:                   64 (bytes)
17  Size of program headers:               56 (bytes)
18  Number of program headers:             13
19  Size of section headers:               64 (bytes)
20  Number of section headers:             31
21  Section header string table index: 30

```

Listing 2.2: Exemplo de uso do `readelf` para analisar o cabeçalho ELF

No Capítulo 3, veremos como implementar nossa própria ferramenta para analisar cabeçalhos ELF e explorar sua estrutura interna.

2.2.8 Considerações de Segurança

O cabeçalho ELF é frequentemente alvo de manipulações em ataques de segurança:

- Alterações maliciosas no campo `e_entry` podem redirecionar a execução para código malicioso.
- Modificações nos campos de offset podem causar interpretação incorreta do arquivo.
- Ajustes nos valores de contagem podem levar a estouro de buffer ou outros problemas.

Estas considerações são relevantes para as aplicações de segurança discutidas no Capítulo 5, particularmente na Seção ?? sobre detecção de malware.

2.2.9 Relação com Outras Partes do ELF

O cabeçalho ELF funciona como ponto de entrada para todas as outras estruturas de dados no arquivo:

- Aponta para a tabela de cabeçalhos de programa, que descreve os segmentos para carregamento (ver Seção 2.3).
- Aponta para a tabela de cabeçalhos de seção, que detalha as seções do arquivo (ver Seção 2.4).
- Define o ponto de entrada para execução, essencial para o carregador do sistema operacional.

Esta estrutura hierárquica permite navegação eficiente pelo arquivo e facilita o carregamento de seus componentes na memória, como veremos nas próximas seções.

2.3 Segmentos ELF (Program Headers)

Os segmentos ELF, definidos na Program Header Table, são essenciais para o carregamento e execução de arquivos ELF pelo sistema operacional. Enquanto o cabeçalho ELF, discutido na Seção 2.2, fornece informações gerais sobre o arquivo, os segmentos descrevem como as partes do arquivo devem ser mapeadas na memória durante a execução.

2.3.1 Função dos Segmentos

Os segmentos representam uma visão do arquivo ELF orientada à execução, descrevendo como o carregador do sistema operacional deve preparar o programa para ser executado:

- Eles mapeiam partes do arquivo para a memória virtual.
- Definem atributos como permissões de leitura, escrita e execução.
- Especificam o alinhamento de memória necessário.
- Podem incluir múltiplas seções com características similares.

Esta abstração é diretamente relacionada ao conceito de memória virtual discutido na Seção 1.2.2, onde o mapeamento entre arquivo e memória é fundamental para a execução eficiente de programas.

2.3.2 Estrutura do Program Header

Cada entrada na Program Header Table é definida pela estrutura `Elf64_Phdr` (para ELF de 64 bits):

```
1 typedef struct {
2     Elf64_Word    p_type;        /* Tipo de segmento */
3     Elf64_Word    p_flags;      /* Flags do segmento */
4     Elf64_Off     p_offset;      /* Offset do segmento no arquivo */
5     Elf64_Addr    p_vaddr;      /* Endereco virtual para carregar o segmento
6     */
7     Elf64_Addr    p_paddr;      /* Endereco fisico (relevante para sistemas
8     embarcados) */
9     Elf64_Xword   p_filesz;      /* Tamanho do segmento no arquivo */
10    Elf64_Xword   p_memsz;      /* Tamanho do segmento na memoria */
11    Elf64_Xword   p_align;      /* Alinhamento do segmento */
12 } Elf64_Phdr;
```

Listing 2.3: Estrutura do Program Header de 64 bits

Esta estrutura permite ao sistema operacional determinar exatamente como cada parte do arquivo deve ser carregada na memória.

2.3.3 Tipos de Segmentos (p_type)

O campo `p_type` especifica o tipo do segmento. Alguns valores comuns incluem:

- `PT_NULL` (0): Entrada de segmento não utilizada
- `PT_LOAD` (1): Segmento carregável
- `PT_DYNAMIC` (2): Informações de ligação dinâmica
- `PT_INTERP` (3): Caminho para o interpretador de programa
- `PT_NOTE` (4): Informações auxiliares
- `PT_PHDR` (6): Entrada para a própria tabela de cabeçalhos de programa
- `PT_TLS` (7): Thread Local Storage

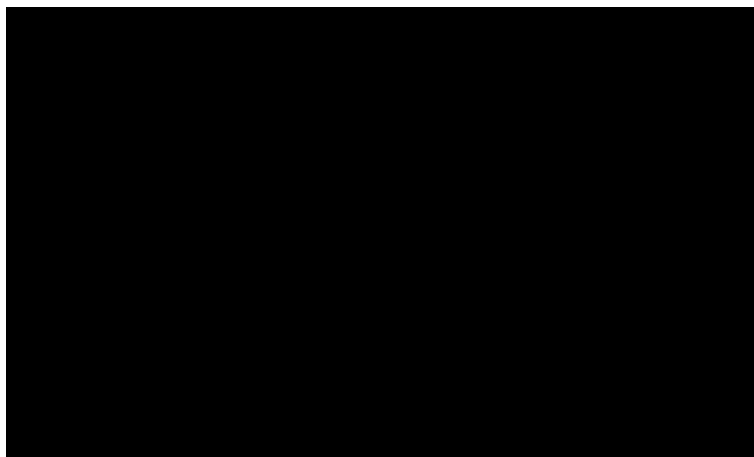


Figure 2.2: Mapeamento de segmentos ELF do arquivo para a memória

A Figura 2.2 ilustra o processo de mapeamento de segmentos do arquivo para a memória. Este processo será explicado em mais detalhes no Capítulo 3 quando discutirmos o carregamento de ELF.

2.3.4 Flags de Segmento (p_flags)

O campo `p_flags` define as permissões do segmento na memória:

- `PF_X` (1): Permissão de execução
- `PF_W` (2): Permissão de escrita
- `PF_R` (4): Permissão de leitura

Estas flags podem ser combinadas para criar diferentes permissões. Por exemplo, um valor de 5 ($= 4 + 1$) representa um segmento com permissão de leitura (R) e execução (X), mas não de escrita (W). Esta configuração é típica para segmentos de código executável, como discutimos na Seção 1.2.3 sobre segmentação de memória.

2.3.5 Mapeamento de Arquivo para Memória

Os campos `p_offset`, `p_vaddr`, `p_filesz`, e `p_memsz` definem como o segmento é mapeado do arquivo para a memória:

- `p_offset`: Posição no arquivo onde o segmento começa
- `p_vaddr`: Endereço virtual onde o segmento deve ser carregado
- `p_filesz`: Tamanho do segmento no arquivo
- `p_memsz`: Tamanho do segmento na memória (pode ser maior que `p_filesz`)

Quando `p_memsz` é maior que `p_filesz`, o sistema operacional preenche a diferença com zeros. Este recurso é utilizado principalmente para o segmento BSS, que contém dados não inicializados.

2.3.6 Segmentos PT_LOAD

Os segmentos do tipo `PT_LOAD` são os mais fundamentais para a execução do programa, pois são os que efetivamente são carregados na memória. Tipicamente, um arquivo ELF executável contém pelo menos dois segmentos `PT_LOAD`:

- Segmento de texto: contém código executável e dados somente leitura (flags: R-X)
- Segmento de dados: contém dados inicializados e espaço para dados não inicializados (flags: RW-)

Esta separação é uma implementação direta do princípio de segmentação discutido anteriormente na Seção 1.2.3.

Table 2.2: Segmentos típicos em um arquivo ELF executável

Tipo	Flags	Conteúdo	Seções Incluídas	Propósito
PT_LOAD	R-X	Código	.text, .rodata	Código executável e cons
PT_LOAD	RW-	Dados	.data, .bss	Variáveis globais
PT_DYNAMIC	RW-	Informações dinâmicas	.dynamic	Ligação dinâmica
PT_INTERP	R-	Caminho do interpretador	.interp	Nome do carregador diná

2.3.7 Segmento PT_INTERP

O segmento PT_INTERP é especialmente importante para programas com ligação dinâmica, pois especifica o caminho para o interpretador de programa (geralmente o carregador dinâmico, como `/lib64/ld-linux-x86-64.so.2`). Este segmento contém uma string terminada em nulo que aponta para o arquivo do interpretador.

Quando o sistema operacional encontra este segmento, ele carrega o interpretador especificado, que então assume o controle para carregar as bibliotecas compartilhadas e resolver símbolos dinâmicos.

2.3.8 Segmento PT_DYNAMIC

O segmento PT_DYNAMIC contém informações cruciais para a ligação dinâmica, incluindo:

- Referências a tabelas de símbolos dinâmicos
- Listas de bibliotecas compartilhadas necessárias
- Informações de relocação
- Endereços de inicialização e finalização

Este segmento é essencial para o funcionamento do carregador dinâmico discutido na Seção [2.3.7](#).

2.3.9 Alinhamento de Segmentos (p_align)

O campo `p_align` especifica o alinhamento necessário para o segmento em memória. Para segmentos carregáveis:

- $p_vaddr \equiv p_offset \pmod{p_align}$
- `p_align` é geralmente uma potência de 2
- Valores típicos são 0x1000 (4KB, tamanho de página padrão)

Este alinhamento é crucial para o desempenho e compatibilidade, conforme discutido na Seção [1.2.4](#).

2.3.10 Análise de Segmentos com Ferramentas

Podemos analisar os segmentos de um arquivo ELF usando ferramentas como `readelf`:

```

1 $ readelf -l /bin/ls
2
3 Elf file type is DYN (Shared object file)
4 Entry point 0x6050
5 There are 13 program headers, starting at offset 64
6
7 Program Headers:
8   Type           Offset             VirtAddr           PhysAddr
9                   FileSiz             MemSiz             Flags   Align
10  PHDR            0x0000000000000040 0x0000000000000040 0x0000000000000040
11                   0x00000000000002d8 0x00000000000002d8  R       0x8
12  INTERP          0x0000000000000318 0x0000000000000318 0x0000000000000318
13                   0x000000000000001c 0x000000000000001c  R       0x1
14    [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
15  LOAD            0x0000000000000000 0x0000000000000000 0x0000000000000000
16                   0x00000000000004d8 0x00000000000004d8  R       0x1000
17  LOAD            0x0000000000000500 0x0000000000000500 0x0000000000000500
18                   0x0000000000001378 0x0000000000001378  R E     0x1000
19  LOAD            0x0000000000001900 0x0000000000001900 0x0000000000001900
20                   0x000000000000053e0 0x000000000000053e0  R       0x1000
21  LOAD            0x0000000000001f00 0x0000000000001f00 0x0000000000001f00
22                   0x0000000000001088 0x0000000000002800  RW      0x1000

```

Listing 2.4: Exemplo de uso do readelf para analisar segmentos

No Capítulo 3, exploraremos como nossa própria ferramenta de análise pode processar e interpretar estas informações.

2.3.11 Relação entre Segmentos e Seções

Como será explicado em maior detalhe na próxima seção (Seção 2.4), existe uma relação importante entre segmentos e seções:

- Seções são unidades lógicas de código ou dados com propósitos específicos.
- Segmentos são unidades de carregamento que podem incluir múltiplas seções.
- A visão por seções é mais relevante para ferramentas como compiladores e linkers.
- A visão por segmentos é mais relevante para o carregamento e execução pelo sistema operacional.

Esta dualidade de perspectivas reflete a natureza dupla do formato ELF como formato tanto para vinculação (linking) quanto para execução.

2.3.12 Considerações para Diferentes Arquiteturas

Embora a estrutura básica dos segmentos ELF seja padronizada, diferentes arquiteturas podem ter requisitos específicos:

- Arquiteturas Harvard puras podem necessitar de segmentos distintos para código e dados.
- Algumas arquiteturas têm requisitos especiais de alinhamento.
- Sistemas embarcados podem usar o campo `p_paddr` (endereço físico) que normalmente é ignorado em sistemas com memória virtual.

Estas considerações serão exploradas mais a fundo na Seção ??, onde discutiremos como o formato ELF se adapta a diferentes arquiteturas de processador.

2.4 Seções ELF (Section Headers)

As seções são as unidades fundamentais de organização em arquivos ELF, fornecendo uma visão detalhada do conteúdo do arquivo para ferramentas de desenvolvimento. Enquanto os segmentos, discutidos na Seção 2.3, são orientados à execução, as seções são orientadas ao processo de vinculação (linking) e manipulação do código.

2.4.1 Conceito e Propósito

As seções dividem o arquivo ELF em unidades lógicas baseadas no tipo de conteúdo e propósito:

- Organizam o código e dados em categorias distintas
- Permitem o acesso seletivo a diferentes partes do arquivo
- Facilitam a manipulação por ferramentas como compiladores, linkers e depuradores
- Fornecem metadados necessários para processamento específico

As seções são particularmente importantes durante o desenvolvimento e para ferramentas de análise, como veremos em maior detalhe no Capítulo 3.

2.4.2 Estrutura do Section Header

Cada seção em um arquivo ELF é descrita por uma entrada na Section Header Table. Para ELF de 64 bits, essa entrada é definida pela estrutura `Elf64_Shdr`:

```
1 typedef struct {
2     Elf64_Word    sh_name;        /* Índice para o nome da seção */
3     Elf64_Word    sh_type;        /* Tipo da seção */
4     Elf64_Xword   sh_flags;       /* Atributos da seção */
5     Elf64_Addr    sh_addr;        /* Endereço virtual, se carregável */
6     Elf64_Off     sh_offset;      /* Offset no arquivo */
7     Elf64_Xword   sh_size;        /* Tamanho da seção em bytes */
8     Elf64_Word    sh_link;        /* Link para outra seção */
9     Elf64_Word    sh_info;        /* Informações adicionais */
10    Elf64_Xword    sh_addralign;   /* Alinhamento necessário */
11    Elf64_Xword    sh_entsize;     /* Tamanho da entrada, se tabela */
12 } Elf64_Shdr;
```

Listing 2.5: Estrutura do Section Header de 64 bits

2.4.3 Tipos de Seção

O campo `sh_type` especifica o tipo da seção, determinando seu conteúdo e propósito:

- `SHT_NULL` (0): Seção inativa ou marcador
- `SHT_PROGBITS` (1): Dados definidos pelo programa
- `SHT_SYMTAB` (2): Tabela de símbolos
- `SHT_STRTAB` (3): Tabela de strings
- `SHT_RELA` (4): Entradas de relocação com adendos explícitos
- `SHT_HASH` (5): Tabela de hash de símbolos
- `SHT_DYNAMIC` (6): Informações para vinculação dinâmica
- `SHT_NOTE` (7): Informações de marcação
- `SHT_NOBITS` (8): Ocupa espaço mas não tem dados no arquivo (ex: BSS)
- `SHT_REL` (9): Entradas de relocação sem adendos explícitos
- `SHT_DYNSYM` (11): Tabela mínima de símbolos para ligação dinâmica

2.4.4 Flags de Seção

O campo `sh_flags` define atributos para a seção:

- `SHF_WRITE` (0x1): Seção contém dados graváveis durante a execução
- `SHF_ALLOC` (0x2): Seção ocupa memória durante a execução
- `SHF_EXECINSTR` (0x4): Seção contém código executável
- `SHF_MERGE` (0x10): Dados podem ser mesclados para economizar espaço
- `SHF_STRINGS` (0x20): Seção contém strings terminadas em nulo
- `SHF_INFO_LINK` (0x40): Campo `sh_info` contém índice de seção
- `SHF_TLS` (0x400): Seção contém dados de Thread-Local Storage

Estas flags podem ser combinadas para definir múltiplos atributos.

2.4.5 Seções Especiais

Certas seções têm significados específicos e padronizados:

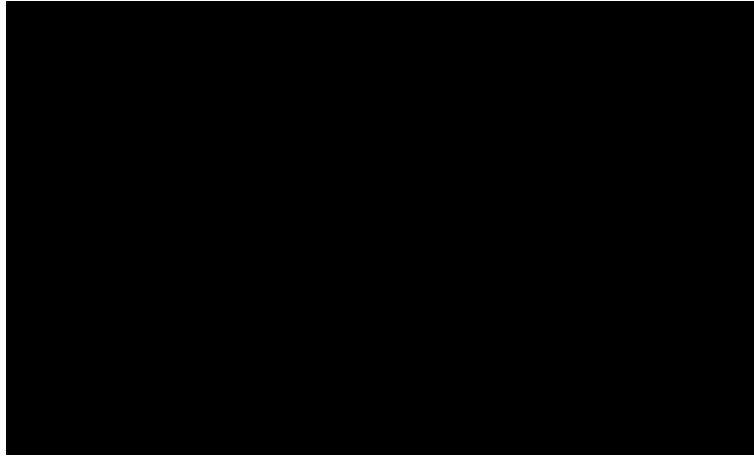


Figure 2.3: Organização das seções em um arquivo ELF típico

Table 2.3: Seções comuns em arquivos ELF

Nome	Tipo	Descrição
.text	SHT_PROGBITS	Código executável do programa
.data	SHT_PROGBITS	Dados inicializados (variáveis globais)
.bss	SHT_NOBITS	Dados não inicializados (ocupa espaço apenas na memória)
.rodata	SHT_PROGBITS	Dados somente para leitura (constantes)
.symtab	SHT_SYMTAB	Tabela de símbolos completa
.strtab	SHT_STRTAB	Tabela de strings para nomes de símbolos
.shstrtab	SHT_STRTAB	Tabela de strings para nomes de seções
.dynamic	SHT_DYNAMIC	Informações para vinculação dinâmica
.got	SHT_PROGBITS	Global Offset Table (para código com posição independente)
.plt	SHT_PROGBITS	Procedure Linkage Table (para chamadas em bibliotecas compartilhadas)
.init	SHT_PROGBITS	Código de inicialização executado antes da função main()
.fini	SHT_PROGBITS	Código de finalização executado após o término do programa

2.4.6 Tabela de Strings da Seção

Os nomes das seções não são armazenados diretamente nos cabeçalhos de seção. Em vez disso, o campo `sh_name` contém um índice para a tabela de strings `.shstrtab`. Esta tabela armazena todos os nomes de seção como strings terminadas em nulo, concatenadas em um único bloco.

O índice para esta tabela de strings é especificado no campo `e_shstrndx` do cabeçalho ELF, como mencionado na Seção 2.2.

2.4.7 Relação com Segmentos

Múltiplas seções podem ser combinadas em um único segmento para carregamento na memória:

- Seções são a visão lógica, orientada à vinculação
- Segmentos são a visão física, orientada à execução

- O mapeamento entre seções e segmentos é definido pela posição e atributos

Por exemplo, as seções `.text`, `.rodata` e outras seções somente leitura seriam tipicamente mapeadas para um segmento `PT_LOAD` com permissões `R-X`, enquanto `.data` e `.bss` seriam mapeadas para um segmento `PT_LOAD` com permissões `RW-`.

2.4.8 Análise de Seções com Ferramentas

As seções de um arquivo ELF podem ser analisadas utilizando ferramentas como `readelf`:

```

1 $ readelf -S /bin/ls
2 There are 29 section headers, starting at offset 0x21768:
3
4 Section Headers:
5   [Nr] Name                Type                Address              Offset
6       Size                EntSize            Flags   Link   Info   Align
7   [ 0]                      NULL               0000000000000000    00000000
8       0000000000000000    0000000000000000                0     0     0
9   [ 1] .interp                PROGBITS           0000000000000318    00000318
10       000000000000001c    0000000000000000      A       0     0     1
11   [ 2] .note.gnu.build-id     NOTE               0000000000000338    00000338
12       0000000000000024    0000000000000000      A       0     0     4
13   ...
14   [11] .text                  PROGBITS           0000000000005000    00005000
15       000000000000efc2    0000000000000000     AX       0     0    16
16   ...

```

Listing 2.6: Exemplo de uso do `readelf` para analisar seções

2.4.9 Seções de Depuração

Arquivos ELF podem incluir seções especiais para informações de depuração:

- `.debug_info`: Informações gerais de depuração
- `.debug_line`: Mapeamento entre código objeto e código fonte
- `.debug_abbrev`: Abreviações para compressão de dados de depuração
- `.debug_str`: Strings utilizadas nas informações de depuração

Estas seções seguem frequentemente o formato DWARF, um padrão para informações de depuração que complementa o ELF. Estas informações são cruciais para depuradores como GDB e para ferramentas de análise que veremos no Capítulo 3.

2.4.10 Uso de Seções em Análise Binária

A análise de seções ELF é particularmente útil em:

- Engenharia reversa: Identificação de código e dados
- Segurança: Detecção de seções suspeitas ou modificadas
- Otimização: Análise de tamanho e organização das seções
- Depuração: Correlação entre código binário e código fonte

Como será discutido no Capítulo 5, estas aplicações são fundamentais para áreas como detecção de malware e patching de software.

2.4.11 Extensões Específicas

Diferentes implementações podem adicionar seções específicas:

- GNU: Seções como `.gnu.version`, `.gnu.hash`
- Linux: Seções específicas do kernel como `.modinfo`
- Compiladores: Seções para otimizações específicas

A flexibilidade do formato ELF permite estas extensões sem comprometer a compatibilidade básica, um dos motivos de sua ampla adoção em sistemas Unix-like.

Chapter 3

Análise de Código

Chapter 4

Análise Comparativa

Chapter 5

Aplicações Futuras

Chapter 6

Conclusão

Chapter 7

Referências