Uma Infra-estrutura Modular e Extensível para a Manipulação de Binários MIPS

por

Ricardo Nabinger Sanchez

UNIVERSIDADE DO VALE DO RIO DOS SINOS

RICARDO NABINGER SANCHEZ

Uma Infra-estrutura Modular e Extensível para a Manipulação de Binários MIPS

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Prof. Dr. César Augusto Missio Marcon
Orientador

São Leopoldo, junho de 2007

LICENÇA DE USO

O conteúdo deste documento é disponibilizado sob a licença revisada BSD[1], na esperança de que possa ser útil. O texto da licença é apresentado na íntegra, a seguir.

[1] Disponível em http://www.opensource.org/licenses/bsd-license.html

"If you think it's simple, then
you have misunderstood the problem."
— Bjarne Stroustrup

AGRADECIMENTOS

Este trabalho só foi possível em função dos esforços coordenados de muitas pessoas. Sendo assim dedico este trabalho a todas elas.

À Débora, pelo total apoio, companheirismo e amor dedicado (e retribuído!). À minha mãe Liane, pelo ambiente que permitiu a realização de um trabalho perigoso em função dos prazos. Ao meu pai Leopoldo, que mesmo falecido há muitos anos, mostrou muitos dos caminhos (inclusive quais são as linguagens de programação utilizadas pelos campeões—C/C++). À minha irmã, que sempre foi uma grande amiga, pelo otimismo. Ao tio Bob, por ter inclusive revisado o relatório de andamento.

Ao meu primo Pedro, por inúmeros conselhos tipográficos (que inclusive ajudaram outros colegas formandos que usam LaTeX), e por aguentar, todo santo dia, todas as bobagens que eu mando pra ele—e são muitas. Ao André, amigo de longuíssima data, por muitos conselhos de aplicação geral—afinal, ele é o mestre. Ao Lucas, que além de aturar *random junk* com muito bom-humor, sempre ofereceu soluções simples para grandes problemas. Ao Hisham, pelos vários esclarecimentos relacionados ao trabalho. Ao FelipeWD, por discutir muitas das idéias antes da definição final do tema, e também pela tarefa do esclarecimento. Ao Rafael, por estar sempre disposto a conversar sobre qualquer coisa, e por ter gentilmente emprestado um AMD Athlon-XP bem potente!

Ao orientador, professor Marcon, por ter, obviamente, ajudado em todas as questões pertinentes ao trabalho, mas em especial por ter ido além, revisando todas as idéias propostas (mesmo as inúteis!) com muito cuidado, e também por ter revisado muitas prévias deste documento. Aos antigos orientadores, professores Luciano e Marinho, pelo conhecimento e experiência adquiridos durante o período de iniciação científica—que foram fundamentais neste trabalho.

É bem provável que eu tenha esquecido de listar outros que contribuíram diretamente com este trabalho—*you know who you are*. Saibam que vocês também tem meu agradecimento. E não deixem de ajudar novamente durante o mestrado! `:-)`

Todos os componentes deste trabalho foram realizados com ferramentas livres e de código aberto. Esta monografia foi feita em LaTeX 2_ε, usando a classe da universidade, a fonte Latin Computer modern, e o BibTeX para as referências bibliográficas. Nenhum dos pacotes LaTeX adicionais foi ferido durante a elaboração da monografia. As ilustrações foram implementadas em Troff, e pré-processadas com o `pic` e `eqn`, sendo que o *driver* utilizado foi o `grodvi`. Todos os arquivos foram editados no *Vim*—o editor dos vencedores—versão 7. O BSD Make evitou muitas recompilações desnecessárias. Subversion (`svn`) foi adotado como o *software* de controle de versão oficial deste trabalho.

SUMÁRIO

LISTA DE ACRÔNIMOS

ASCII *American Standard Code for Information Interchange*

BSD *Berkeley Software Distribution*

CFG *Control Flow Graph*

DFG *Data Flow Graph*

ELF *Executable and Linking Format*

GCC *GNU Compiler Collection*

GNU *GNU is Not UNIX*

GPL *General Public License*

MIPS *Microprocessor without Interlocked Pipeline Stages*

RISC *Reduced Instruction-Set Computer*

RTL *Register Transfer Language*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

RESUMO

Desde o final da década de 1950, pesquisadores vêm investindo muitos recursos no estudo e desenvolvimento de técnicas de otimização, e recentemente em técnicas de re-engenharia de código, com o objetivo de gerar ou adaptar código de forma automática e que seja capaz de obter o melhor aproveitamento dos recursos disponíveis. Com a atual diversidade de arquiteturas, sistemas operacionais, linguagens de programação, e compiladores, soluções envolvendo interpretação de código e tradução binária têm se mostrado cada vez mais efetivas e com amplo espectro de aplicação.

Este trabalho propõe uma infra-estrutura de *software* modular e extensível para a manipulação de código executável. O fluxo de operação é organizado em estágios, onde cada estágio é composto por um ou mais módulos, sendo que cada módulo implementa uma funcionalidade específica, como por exemplo a aplicação de uma técnica de otimização. A infra-estrutura proposta foi implementada, dando origem à ferramenta ReMIPS. Como estudo de caso, a ferramenta ReMIPS foi implementada de forma a suportar o formato ELF de arquivos-objeto, padrão *de facto* em sistemas operacionais UNIX e semelhantes ao UNIX, e também a arquitetura MIPS de microprocessadores RISC. ReMIPS contempla a abordagem modular proposta, sendo capaz de aplicar uma técnica de otimização diretamente a um binário executável, validando a efetividade da solução proposta.

Palavras-chave: Re-engenharia de código. Otimização de código. Engenharia reversa. MIPS.

ReMIPS: a Modular and Extensible Infra-structure for Optimizing MIPS Binaries

ABSTRACT

Since the late 1950's, researchers have been investing a lot of resources in the study and development of optimization techniques, and recently in code re-engineering techniques, aiming at better code generation in an automatic fashion and capable of making the best use of the available resources. With today's diversity of architectures, operating systems, programming languages, and compilers, solutions involving code interpretation and binary translation have shown themselves more and more effectives and amply applicable.

This work proposes a modular and extensible software infra-structure for the manipulation of executable code. The workflow is organized in stages, where each stage is comprised by one or more modules, and each module implements a well-defined task, for instance applying an optimization technique. The proposed infra-structure was implemented, originating a tool called ReMIPS. As a case study, the ReMIPS tool was implemented in such a way to support the ELF object-file format, a *de facto* standard in UNIX and UNIX-like operating systems, and also the MIPS architecture of RISC microprocessors. ReMIPS fulfil the modular approach proposed, being capable of applying an optimization technique directly to an executable binary, supporting the effectiveness of the proposed solution.

Keywords: Code re-engineering. Code optimization. Reverse engineering. MIPS.

1 INTRODUÇÃO

No final da década de 1950, a programação dos computadores era um processo lento e propenso a erros, realizada apenas por programadores experientes que usavam linguagem de montagem ou de máquina. Para contornar estas limitações, pesquisadores introduziram linguagens de programação de alto nível e, conseqüentemente, os primeiros compiladores capazes de realizar a "programação automática" [1]. Parte do sucesso destas linguagens se deve à qualidade e ao volume de recursos investidos na pesquisa e desenvolvimento de técnicas de otimização de código, permitindo que os compiladores gerassem código-objeto que executava com eficiência comparável ao gerado por programadores experientes.

Os avanços tecnológicos promovidos pelos fabricantes de processadores são acompanhados por diversos grupos de pesquisa, que buscam um aproveitamento cada vez maior dos recursos computacionais disponíveis a partir do refinamento e desenvolvimento de novas técnicas de otimização. No contexto deste trabalho, *técnicas de otimização*, ou simplesmente *otimização*, diz respeito às alterações efetuadas no fluxo de instruções a fim de se obter uma computação equivalente, mas com a utilização de menos recursos, como tempo, memória ou energia. Parte deste esforço visa avaliar os ganhos obtidos com as técnicas existentes, bem como verificar a interação entre as mesmas, especialmente para identificar resultados negativos que possam surgir destas interações [2; 3].

1.1 Motivação

Compiladores como o GCC[1] [4] implementam diversas técnicas de otimização para as arquiteturas-alvo suportadas, tanto genéricas quanto específicas. No contexto deste trabalho, arquitetura-alvo é a combinação de formato de arquivo-objeto, sistema operacional, e processador para os quais um binário executável é gerado, como por exemplo ELF-FreeBSD/MIPS, que denota o formato ELF[2] de arquivos-objeto, gerado para o sistema operacional FreeBSD executando sobre um processador da arquitetura MIPS[3]. Técnicas específicas de otimização são aquelas que tiram proveito de detalhes da arquitetura-alvo, como por exemplo o *escalonamento de instruções*. Por outro lado, técnicas genéricas de otimização atuam no nível do código fonte, como por exemplo a *remoção de código inatingível*[4] [5], podendo ser aplicadas independentemente da arquitetura-alvo subjacente. Algumas técnicas de otimização

[1] *GNU Compiler Collection*
[2] *Executable and Linking Format*
[3] *Microprocessor without Interlocked Pipeline Stages*
[4] *Unused code removal*, em Inglês.

são aplicáveis apenas em determinados contextos, como por exemplo a *expansão de laços*[5] [5; 6], que realiza transformações no nível do código fonte levando em consideração características da arquitetura-alvo.

No caso específico das versões anteriores a 4 do GCC, o estágio de otimização é composto por diversas etapas, sendo que cada uma delas implementa uma técnica de otimização específica. Por razões históricas, a ordem de aplicação das técnicas de otimização é fixa, o que sob certas condições pode resultar em código de qualidade inferior. Contudo, a partir da versão 4 do compilador o estágio de otimização foi remodelado, reduzindo consideravelmente a incidência de efeitos colaterais entre as técnicas de otimização [7], além de permitir alterações na ordem de aplicação das técnicas. Sendo assim, não basta aos desenvolvedores e pesquisadores proporem novas técnicas de otimização sem considerar os possíveis efeitos, tanto positivos quanto negativos, resultantes da interação com as demais técnicas.

1.2 Trabalhos Relacionados

Este trabalho está inserido nos contextos de *desmontagem de binários* e *re-engenharia de código*. O primeiro diz respeito à decodificação de instruções de máquina presentes em binários executáveis, enquanto que o segundo engloba diversas questões relacionadas à manipulação de código, inclusive as *técnicas de otimização*. Estas podem ser consideradas parte do contexto de re-engenharia de código, porém voltadas à redução de recursos necessários para realizar uma computação, em termos de tempo, energia ou memória.

A desmontagem de binários compreende tanto abordagens *estáticas* quanto *dinâmicas* [8]. Nas abordagens estáticas um desmontador analisa o conteúdo de um binário executável sem executá-lo [6; 9; 10; 11; 12], isto é, decodifica as instruções sem manter um contexto de execução real. Devido às limitações intrínsecas das abordagens estáticas, algumas incluem formas de interpretar ou executar simbolicamente as instruções que julgarem relevantes para a desmontagem [13; 14; 15], apesar de a abordagem em si permanecer estática. Por outro lado, nas abordagens dinâmicas o binário é executado de forma assistida ou monitorada, sendo que trechos do código são desmontados de acordo com a execução real do binário [16; 17; 18], fazendo com que a desmontagem em si dependa da entrada fornecida ao programa, pois em geral apenas os trechos efetivamente executados são desmontados.

De forma semelhante às técnicas de desmontagem, as técnicas relacionadas à re-engenharia de código podem ser aplicadas antes ou depois da geração de código. No primeiro caso, transformações são realizadas sobre uma representação intermediária do compilador anterior à geração de código [19; 20; 21], ou sobre código em linguagem de montagem[6] [22]. As abordagens aplicáveis após a geração de código têm sido bastante exploradas em trabalhos que realizam tradução binária [23; 24; 25; 26], um processo usado para converter instruções de um conjunto de instruções para outro, preservando sua semântica, e também por infra-estruturas de otimização em tempo de execução [16; 17; 26; 27]. Tanto nas abordagens anteriores quanto nas posteriores à geração de código por um compilador ou montador, é bastante comum o emprego da re-engenharia de código visando otimizá-lo para obter ganhos específicos, como

[5] *Loop-unrolling*, em Inglês.

[6] *Assembly Language* (em Inglês) é uma classe de linguagens de baixo nível, cujo processamento é feito por um *montador* (*assembler*), gerando instruções de máquina.

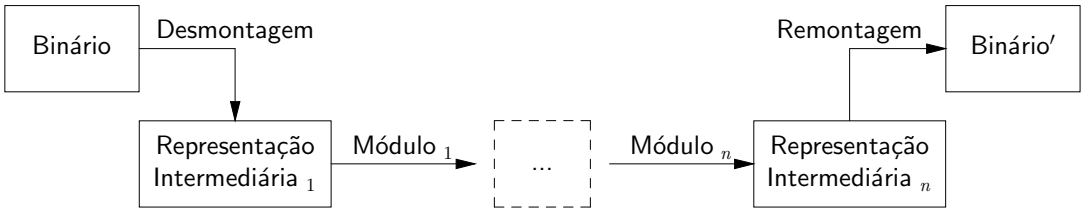

Figura 1.1: Visão geral da infra-estrutura proposta. Um arquivo-objeto é desmontado em uma representação intermediária manipulável, que é sucessivamente analisada e modificada por módulos de alteração binária. Ao final do processo, a representação intermediária resultante é remontada em um novo arquivo-objeto de saída.

por exemplo um menor tempo de execução, utilização de memória ou consumo de energia.

Alguns trabalhos propõem soluções para facilitar o estudo e desenvolvimento de novas técnicas de otimização [28; 29], ou ainda abordagens alternativas à manipulação de instruções, como a edição binária [30; 31; 32], uma forma de re-engenharia aplicada ao arquivo-objeto em si. Em ambos os casos, os autores apresentam resultados muito significativos.

1.3 Objetivos

Este trabalho propõe uma infra-estrutura de *software* extensível para a manipulação de binários executáveis a partir da aplicação de técnicas de re-engenharia de código. A principal motivação para manipular diretamente código binário é a generalização de soluções para binários gerados a partir de diferentes linguagens de programação de alto-nível [12]. Ainda, é possível realizar otimizações com resultados bastante significativos [6; 12; 16; 17; 18; 27], apesar do baixo nível de abstração. Com exceção da infra-estrutura em si, as funcionalidades são implementadas através de módulos, sendo possível expandir o espectro de aplicação da solução proposta a partir da implementação de módulos adicionais. Como estudo de caso, adotou-se o formato ELF de arquivos-objeto, padrão *de facto* para a plataforma UNIX, e a arquitetura MIPS, por ser do tipo RISC[7], ou seja, uma arquitetura com um conjunto reduzido de instruções.

A Figura 1.1 apresenta uma visão geral da infra-estrutura proposta. Os dados de entrada, geralmente um arquivo-objeto, são analisados em busca de instruções de máquina, que são armazenadas em uma representação intermediária. Esta representação intermediária é fornecida a cada um dos n módulos, que podem modificá-la de acordo com seus objetivos, como por exemplo aplicar alguma técnica de otimização. Após a aplicação do último módulo, o estado final da representação intermediária é remontado em um novo arquivo-objeto, um processo inverso ao de desmontagem realizado no início de todo o processo.

O restante deste trabalho está organizado da seguinte forma. O Capítulo 2 detalha as características do formato de arquivos-objeto ELF, como sua organização interna, estruturas de dados, e também uma visão geral de sua interpretação. Algumas das técnicas para extração de instruções, um processo denominado *des-*

[7] *Reduced Instruction-Set Computer*

montagem, são contextualizadas e comparadas no Capítulo 3.

O Capítulo 4 apresenta o formato interno de representação empregado na infra-estrutura. Em função de algumas limitações das técnicas tradicionais de desmontagem, uma extensão à uma das técnicas é apresentada no Capítulo 5, bem como os problemas que se busca contornar a partir de sua utilização.

A extensibilidade da proposta a partir da modularização é discutida no Capítulo 6, que apresenta os detalhes internos da infra-estrutura proposta, bem como a relação existente entre os diversos módulos. O Capítulo 7 apresenta os detalhes da ferramenta ReMIPS, implementada como estudo de caso e validação da infra-estrutura proposta, bem como os resultados obtidos. Finalmente, o Capítulo 8 apresenta as considerações finais e trabalhos futuros, encerrando este trabalho.

2 ARQUIVOS-OBJETO ELF

Arquivos-objetos são arquivos de uso especial que armazenam código executável, dados, e informações simbólicas geradas por um compilador a partir de um código fonte [33]. ELF, *Executable and Linking Format* em Inglês, é um formato de arquivo-objeto apropriado para vários tipos de arquivos, como binários executáveis (programas), bibliotecas compartilhadas, entre outros [34]. Sua organização interna se dá por meio de *segmentos*, descritos por *cabeçalhos secundários*, que por sua vez são descritos por um *cabeçalho principal*. Esta organização confere aos arquivos ELF uma grande flexibilidade, sem comprometer a eficiência de acesso já que qualquer segmento pode ser facilmente localizado.

Devido às suas características, muitas das versões disponíveis de sistemas operacionais UNIX ([35; 36]) e semelhantes ao UNIX ([37; 38]) adotam, ou pelo menos suportam o formato ELF, o que o torna um padrão *de facto* para as plataformas UNIX existentes.

Este Capítulo está organizado da seguinte forma: a estrutura interna de um arquivo ELF é apresentada na Seção 2.1, seguida por considerações a respeito de diferentes representações de palavras na memória na Seção 2.2. A Seção 2.3 encerra o Capítulo com considerações a respeito da interpretação de arquivos ELF.

2.1 Estrutura Interna

Um arquivo ELF é composto por um *cabeçalho principal*, *cabeçalhos de programa*, *cabeçalhos de seção*, e *segmentos de dados*. Cada um destes componentes é detalhado nas Subseções a seguir. A Figura 2.1 ilustra a disposição interna como sugerido em [34]. Vale lembrar que o leiaute ilustrado não é mandatório, embora seja muito comum encontrar ferramentas que o seguem.

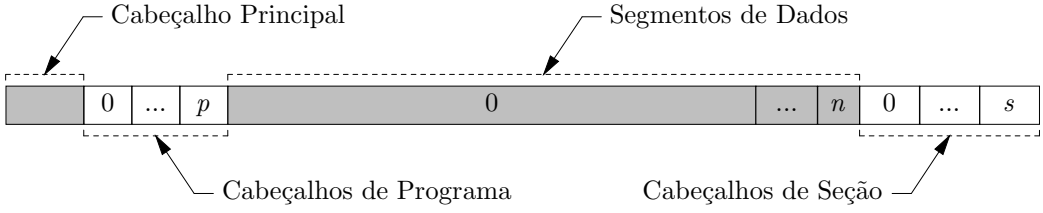

Figura 2.1: Disposição típica de cabeçalhos e segmentos em arquivos-objeto ELF.

Tabela 2.1: Descrição dos atributos do cabeçalho principal de arquivos ELF.

Atributo	Conteúdo
`e_ident`	Seqüência de bytes contendo diversos parâmetros de identificação necessários para a correta interpretação do arquivo.
`e_type`	Tipo do arquivo, indicando se o arquivo é relocável, executável, compartilhado (biblioteca) ou de núcleo (*core*).
`e_machine`	Arquitetura-alvo do arquivo.
`e_version`	Especificação ELF que o arquivo atende.
`e_entry`	Endereço virtual do ponto de entrada do programa, caso o arquivo seja um binário executável.
`e_phoff`	Posição absoluta no arquivo onde se encontra o conjunto de cabeçalhos de programa.
`e_shoff`	Posição absoluta no arquivo onde se encontra o conjunto de cabeçalhos de seção.
`e_flags`	Atributos específicos para processadores, não definidos pela especificação.
`e_ehsize`	Tamanho (em bytes) do cabeçalho ELF.
`e_phentsize`	Tamanho (em bytes) de cada cabeçalho de programa.
`e_phnum`	Quantidade de cabeçalhos de programa.
`e_shentsize`	Tamanho (em bytes) de cada cabeçalho de seção.
`e_shnum`	Quantidade de cabeçalhos de seção.
`e_shstrndx`	Especifica qual cabeçalho de seção indica o segmento contendo uma tabela de *strings* com o nome de cada seção presente no arquivo.

2.1.1 Cabeçalho Principal

Localizado exatamente no início do arquivo, o cabeçalho principal concentra diversas informações necessárias para a interpretação do arquivo, como por exemplo arquitetura-alvo e ordem de bytes adotada. Além disso, contém informações a respeito dos cabeçalhos secundários, como quantidade, tamanho e localização. A Tabela 2.1 apresenta os atributos presentes em um cabeçalho ELF com um breve esclarecimento de cada um.

Para garantir que o formato fosse usado por vários anos sem modificações significativas, o padrão contém detalhes de implementação para arquiteturas tanto de 32 quanto de 64 bits. Ainda, a apresentação do atributo `e_ident` como seqüência de bytes, e não palavras, foi deliberadamente escolhida para permitir que um arquivo ELF possa ser minimamente analisado em ambientes que suportem o formato ELF, mas possuam características diferentes das especificadas no arquivo.

Com as informações presentes no cabeçalho principal, as ferramentas, bem como o sistema operacional, podem decidir se têm ou não condições de interpretar o arquivo.

Tabela 2.2: Descrição dos atributos dos cabeçalhos de seção.

Atributo	Conteúdo
`sh_name`	Índice na tabela de nomes de seção, indicando o nome da seção correspondente ao cabeçalho.
`sh_type`	Indica o tipo de seção e semântica associada ao conteúdo.
`sh_flags`	Propriedades adicionais da seção, como permissão de leitura ou escrita, ou ainda presença de instruções executáveis.
`sh_addr`	Endereço virtual onde o conteúdo da seção deve ser carregado.
`sh_offset`	Posição absoluta do conteúdo da seção no arquivo.
`sh_size`	Tamanho do conteúdo da seção (bytes).
`sh_link`	Elo para outro cabeçalho de seção, cuja interpretação depende do tipo de seção.
`sh_info`	Informações adicionais, cuja interpretação depende do tipo de seção.
`sh_addralign`	Alinhamento do conteúdo tanto em disco quanto em memória, expressado como expoente para base 2.
`sh_entsize`	Tamanho de cada entrada (em bytes), caso seja relevante para a seção, como ocorre em seções contendo tabela de símbolos.

2.1.2 Cabeçalhos de Seção

Descrevem segmentos no arquivo, indicando, por exemplo, a localização em disco, tamanho, tipo de segmento, além de outras informações. Seus atributos são detalhados na Tabela 2.2.

No arquivo, os cabeçalhos de seção são dispostos seqüencialmente como um vetor, sendo que todos os elementos possuem o mesmo tamanho, conforme especificado no atributo `e_shentsize` do cabeçalho principal. Alguns índices no vetor de cabeçalhos são reservados, pois possuem significado especial, e são apresentados na Tabela 2.3. As seções reservadas nunca devem estar presentes no arquivo, apenas referências a elas a partir do atributo `sh_link` no cabeçalho de seção.

A quantidade, tamanho unitário dos elementos, e localização no arquivo são especificados no cabeçalho principal, e com elas pode-se localizar rapidamente a posição exata de qualquer cabeçalho de seção no arquivo. A posição em arquivo do i-ésimo cabeçalho de seção é dada por:

$$\texttt{e\_shoff} + i \times \texttt{e\_shentsize}, \quad 0 \leq i < \texttt{e\_shnum}$$

2.1.3 Cabeçalhos de Programa

Descrevem segmentos no arquivo, de forma análoga aos cabeçalhos de seção, mas foram projetados para serem usados pelo sistema operacional durante a carga de um programa ou biblioteca para a memória principal, antes da execução. Um segmento descrito por um cabeçalho de programa não precisa necessariamente coincidir com um ou mais segmentos descritos por cabeçalhos de seção. Por esta razão, é possível que um único cabeçalho de programa seja usado para descrever todos os segmentos executáveis de um arquivo ELF, desde que sejam adjacentes tanto em disco quanto em memória, permitindo que a carga seja bastante eficiente.

Tabela 2.3: Seções com semântica especial em arquivos ELF. A primeira coluna indica o índice da seção reservada, a segunda apresenta o valor simbólico atribuído, e a última apresenta a semântica associada a tais seções.

Índice	Identificador	Significado
0	SHN_UNDEF	Seção nula.
0xff00	SHN_LORESERVE	Limite inferior das demais seções reservadas.
0xff00	SHN_LOPROC	Limite inferior dos índices reservados específicos do processador.
0xff1f	SHN_HIPROC	Limite superior dos índices reservados para processadores.
0xfff1	SHN_ABS	Informações presentes em representação absoluta, e que não devem ser afetadas por relocação.
0xfff2	SHN_COMMON	Referências a símbolos externos não presentes no arquivo.
0xffff	SHN_HIRESERVE	Limite superior dos índices reservados.

Assim como os cabeçalhos de seção, os cabeçalhos de programa também são organizados em disco sob a forma de um vetor. A quantidade de cabeçalhos é indicada pelo atributo `e_phnum` do cabeçalho principal, bem como o tamanho unitário (`e_phentsize`) e a posição em disco do primeiro elemento (`e_phoff`). A posição em disco do j-ésimo elemento pode ser determinada de forma semelhante aos cabeçalhos de seção, usando-se apenas informações presentes no cabeçalho principal:

$$\texttt{e\_phoff} + j \times \texttt{e\_phentsize}, \quad 0 \leq j < \texttt{e\_phnum}$$

A Tabela 2.4 descreve os atributos presentes nos cabeçalhos de programa de arquivos ELF.

2.1.4 Segmentos de Dados

Um segmento de dados é basicamente um bloco composto por uma seqüência de bytes no arquivo. Em geral, um arquivo ELF é composto por múltiplos segmentos, cada um com sua própria semântica, que podem ser descritos tanto por cabeçalhos de seção quanto de programa. Alguns tipos de segmentos, comuns a diversos métodos de geração de código, são definidos pela especificação ELF ([34]), enquanto que outras ficam a critério do gerador de código.

Como exemplo, os segmentos `.dynamic`, `.hash`, `.strtab` e `.symtab` deverão conter, respectivamente, informações para ligação dinâmica, tabela *hash* de símbolos, tabela contendo nomes de símbolos, e tabela de símbolos. Ainda, a especificação ELF define as estruturas de dados que devem ser adotadas para popular estes segmentos, bem como as formas de acesso, o que é especialmente importante para a seção `.hash`.

Outros segmentos possuem semântica pré-definida, mas seu conteúdo fica a critério da ferramenta usada (compilador, por exemplo), em função da arquitetura-alvo ou sistema operacional. Os segmentos `.data`, `.debug` e `.text` deverão conter, respectivamente, dados inicializados que farão parte da imagem em memória do programa, informações simbólicas para depuração, e instruções de máquina. Contudo,

Tabela 2.4: Descrição dos atributos dos cabeçalhos de programa.

Atributo	Conteúdo
`p_type`	Tipo de segmento que o cabeçalho descreve
`p_offset`	Posição absoluta do segmento no arquivo
`p_vaddr`	Endereço virtual de memória onde o segmento deve ser carregado
`p_paddr`	Endereço físico de memória, se relevante ou suportado pelo sistema operacional
`p_filesz`	Tamanho do segmento em disco (bytes)
`p_memsz`	Tamanho do segmento em memória (bytes)
`p_flags`	Propriedades que indicam se o segmento é executável, pode ser lido, pode ser modificado, ou alguma combinação
`p_align`	Alinhamento do segmento tanto em disco quanto em memória, expressado como expoente para base 2

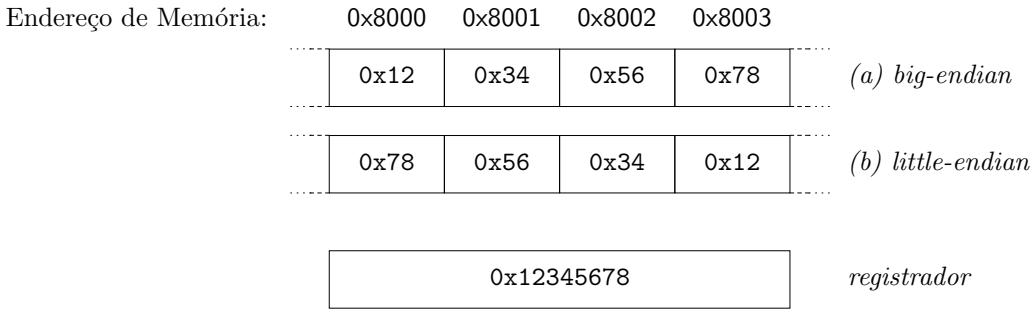

Figura 2.2: Diferenças da representação em memória entre *big-* e *little-endian* para a palavra `0x12345678`. A parte superior ilustra a disposição de cada byte da palavra nos bytes de memória que ela ocupa, contrastando com a representação em um registrador do processador, na parte inferior da Figura.

ao contrário dos segmentos citados anteriormente, a especificação define apenas a semântica dos dados, mas não seu formato ou forma de acesso, ficando a critério do gerador de código.

2.2 Ordem de Bytes

A especificação do formato ELF prevê diferenças entre arquiteturas, tal como a ordem de bytes utilizada pelo processador. Atualmente, existem duas amplamente utilizadas, ilustradas na Figura 2.2: (a) *big-endian*, onde o byte *mais* significativo de uma palavra encontra-se no menor endereço de memória; e (b) *little-endian*, onde o byte *menos* significativo encontra-se no menor endereço de memória [39]. A parte inferior da Figura ilustra o conteúdo de um registrador do processador após receber o conteúdo da palavra em memória.

A arquitetura MIPS, adotada como estudo de caso para este trabalho, geralmente

está associada à ordem de bytes *big-endian*. Contudo, alguns fabricantes produzem processadores MIPS *little-endian* e também *bi-endian*[1], isto é, capazes de alternar entre *little-* e *big-endian* a partir de uma instrução específica.

Para garantir que o mesmo formato de arquivo possa ser utilizado independentemente destas diferenças, um atributo especial no cabeçalho principal foi reservado para indicar a ordem de bytes (*endianess*) adotada no arquivo. Os primeiros 16 bytes do arquivo ELF compõem o atributo `e_ident` do cabeçalho principal sob a forma de um vetor de bytes, que é, portanto, neutro às diferentes ordens de bytes. Neste vetor, o sexto elemento, denotado simbolicamente pelo índice `EI_DATA`, indica a ordem de bytes adotada em todo o restante do arquivo.

Esta informação é essencial para a correta interpretação do arquivo ELF.

2.3 Interpretação

Uma representação resumida do processo de interpretação de um arquivo ELF é apresentada no Algoritmo 2.1, e detalhado a seguir.

Algoritmo 2.1 Interpretação de um arquivo-objeto ELF

```
 1: E ← ler_cabecalho_elf(arquivo)
 2: if endianess(E) ≠ endianess(M) then
 3:     E ← corrige_endianess(E, endianess(M))
 4: S ← ler_cabecalhos_secao(E, arquivo)
 5: if endianess(E) ≠ endianess(M) then
 6:     for i = 1 to secoes(E) do
 7:         S_i ← corrige_endianess(S_i, endianess(M))
 8: for i = 1 to secoes(E) do
 9:     if contem_instrucoes(S_i) then
10:         D_i ← ler_segmento(S_i, arquivo)
11:         if endianess(E) ≠ endianess(M) then
12:             D_i ← corrige_endianess(D_i, endianess(M))
```

Linhas 1–3 A interpretação de um arquivo ELF inicia com a leitura do cabeçalho principal, localizado em uma posição pré-determinada no arquivo (linha 1). Para a correta interpretação do cabeçalho principal, é necessário verificar a ordem de bytes do mesmo, especificada no próprio cabeçalho, no atributo `e_ident`. Por este atributo não ser constituído de palavras, e sim por bytes individuais, ele não é afetado pela ordem de bytes adotada no arquivo, podendo ser avaliado independentemente da arquitetura (linha 2), como descrito na Seção 2.2. Após a verificação, a ferramenta decide se deve ou não converter as palavras do cabeçalho principal para a ordem de bytes suportada pela arquitetura hospedeira[2] (linhas 2–3).

[1] As denominações *bi-* e *dual-endian* são equivalentes, porém esta última é usada com menor freqüência.

[2] *Arquitetura hospedeira*, no contexto deste trabalho, é a arquitetura na qual a ferramenta implementada se encontra em execução.

Linhas 4–7 Com o cabeçalho principal, a posição e quantidade de cabeçalhos de seção podem ser determinadas. De forma análoga ao cabeçalho principal, os cabeçalhos de seção são lidos do arquivo em disco (linha 4) e, se necessário, têm a ordem de bytes de seus atributos convertidos para a ordem suportada pela arquitetura (linhas 5–7).

Linhas 8–12 Para decidir quais segmentos de dados a ferramenta deve ler do arquivo, ela analisa todos os cabeçalhos de seção em busca do indicador `SHF_EXECINSTR` (linhas 8–9). Caso a seção esteja marcada como tendo instruções executáveis, seu respectivo segmento é carregado para processamento posterior. Analogamente aos cabeçalhos principal e de seção, se necessário o conteúdo dos segmentos têm sua ordem de bytes convertidas para a suportada pela arquitetura-hospedeira.

A Seção 7.1.1 apresenta detalhadamente o processos de identificação de arquivos-objeto ELF, até o momento em que seu conteúdo possa ser processado pelos próximos estágios.

3 DESMONTAGEM

Logo após a interpretação de um arquivo-objeto ELF, tem início a fase denominada de *desmontagem*, que tem por objetivo identificar instruções executáveis para que possam ser decodificadas e analisadas.

As diversas técnicas de desmontagem podem ser agrupadas a partir de seu funcionamento básico, para melhor entendimento. Este Capítulo inicia apresentando a técnica de desmontagem *linear*, na Seção 3.1, seguida pela técnica *recursiva*, na Seção 3.2, e então pela técnica *especulativa*, na Seção 3.3. O Capítulo encerra com considerações a respeito da decodificação de instruções, um processo que geralmente ocorre concomitantemente à desmontagem, na Seção 3.4.

3.1 Desmontagem Linear

É a mais simples das técnicas de desmontagem existentes, bem como a mais eficiente e de mais fácil implementação. Na bibliografia, ela também é conhecida por *varredura linear* (*linear sweep*, em Inglês) [8; 10; 13].

A premissa fundamental desta técnica é que um determinado bloco é constituído exclusivamente por instruções. Neste contexto, a desmontagem consiste apenas no percorrimento seqüencial de todos os elementos de um vetor contendo n elementos (instruções). Em arquiteturas que possuem instruções de tamanho variável, como é o caso da arquitetura Intel `80x86`, o desmontador precisa decodificar minimamente a instrução para determinar a localização da próxima. Por outro lado, a arquitetura MIPS possui instruções de tamanho fixo, o que torna a desmontagem linear trivial neste caso.

Devido à sua simplicidade, um desmontador que adote esta técnica não será capaz de distinguir entre dados e instruções, eventualmente decodificando incorretamente dados como sendo instruções [8; 10; 13]. Alguns compiladores incluem dados no fluxo de instruções para modelar tabelas de saltos (*jump tables* em Inglês), uma construção bastante usada na implementação de condicionais múltiplas (`switch` da linguagem C) [40]. A Figura 3.1 ilustra esta situação a partir do trecho de código em linguagem C em (a), que pode ser implementado em linguagem de montagem MIPS como apresentado em (b). Inicialmente, o endereço base da tabela de saltos (como ressaltado em (b)), é carregado na instrução 1, previamente atribuído ao registrador `$a0` para este exemplo. Em seguida, o resultado da condição computada, devidamente truncado para o tamanho da tabela, é adicionado ao endereço base da mesma, para determinar a localização do i-ésimo elemento da tabela. Finalmente o endereço efetivo é carregado para um registrador (instrução 3), e o desvio é feito na instrução 4. Por assumir que o bloco (b) contém apenas instruções, um desmontador

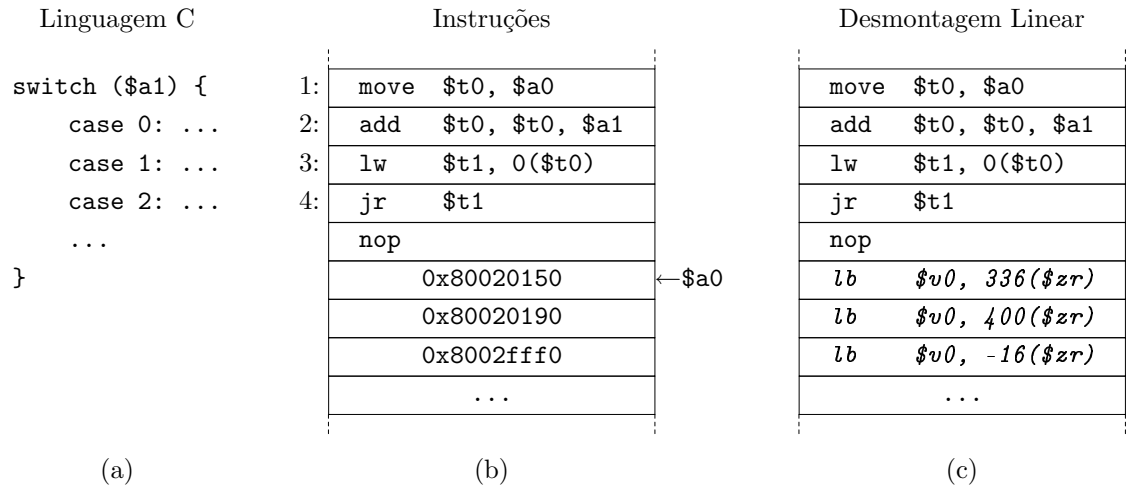

Figura 3.1: A construção `switch` da linguagem `C` em (a) pode ser implementado através da tabela de saltos em (b). Devido à presença de dados no final do fluxo de instruções, o bloco (b) é desmontado incorretamente com a técnica linear gerando o código (c). Neste caso, o resultado da desmontagem contém instruções que não existem no código original.

linear processa os dados presentes no bloco como se fossem instruções, resultando na desmontagem parcialmente incorreta ilustrada em (c), a partir da instrução `nop`. A menos que o desmontador identifique uma instrução inválida no fluxo de instruções, ele não terá como distinguir entre dados e instruções.

Contudo, esta limitação pode ser amenizada com o cruzamento de informações simbólicas, presentes nas tabelas de símbolos e de relocação existentes em diversos formatos de arquivos-objeto, permitindo que o desmontador identifique dados no fluxo de instruções, como é o caso de [30].

3.2 Desmontagem Recursiva

A característica fundamental desta técnica é desmontar instruções de forma semelhante à execução real pelo processador, percorrendo as instruções de acordo com o fluxo de controle, e permitindo a modelagem de um grafo de dependência de controle (CFG[1]) naturalmente.

Esta técnica depende da correta identificação de instruções de desvio, que podem ser *incondicionais*, quando existe apenas um destino possível, ou *condicionais*, quando existem dois destinos possíveis. Para desvios condicionais, diz-se que ele é *tomado*, quando a condição do desvio é *verdadeira* e a execução (ou desmontagem) continua na localização indicada, ou *não-tomado*, quando a condição é *falsa* e a execução segue na próxima instrução, que é, portando o destino *implícito*.

Na arquitetura MIPS, a próxima instrução a um desvio x é a instrução $x + 2$, pois ela emprega desvios atrasados (*delayed branches*, em Inglês) [41; 42]. Por causa disto, sempre que um processador MIPS se depara com uma instrução de desvio, a instrução imediatamente após ao desvio (conhecida como *delay slot*) é executada enquanto o destino do desvio é computado, independentemente do tipo de desvio.

[1] *Control Flow Graph*

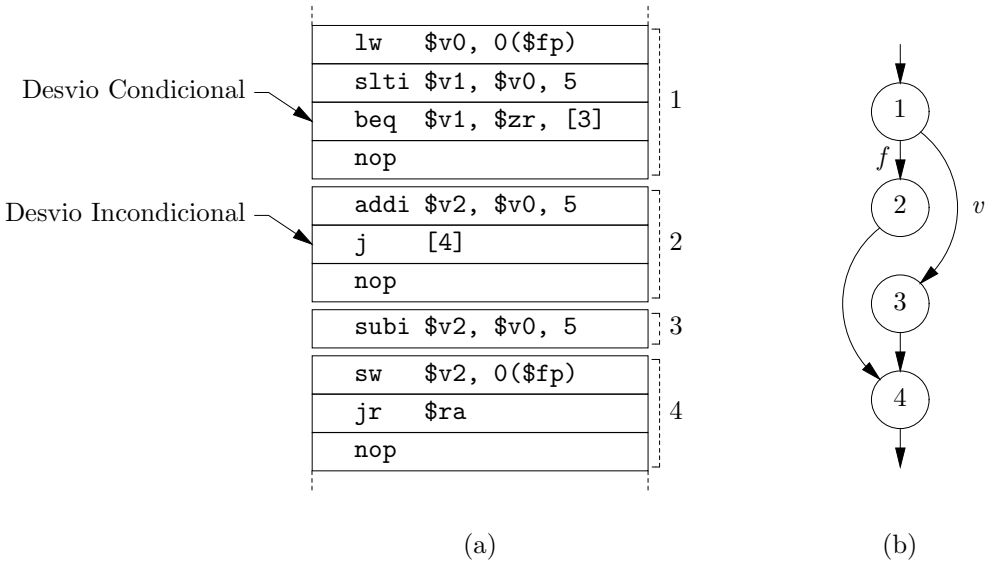

Figura 3.2: Bloco de instruções exemplificando desvios condicional no nodo 1 e incondicional nos nodos 2 e 4. O nodo 3 constitui um caso especial, detalhado no Capítulo 4.

Ainda, o estado do processador é imprevisível caso uma instrução de controle de fluxo seja colocada no *delay slot*.

A Figura 3.2 ilustra em (a) um trecho de instruções que contém as situações recém mencionadas, estruturando-as em (b) sob a forma de um grafo de fluxo de controle (estes grafos são detalhados no Capítulo 4). O nodo 1 apresenta uma instrução de desvio condicional, e caso a condição `$v1` = `$zr` seja verdadeira, a execução (ou desmontagem) continua no nodo 3 (ramo v na Figura 3.2 (b)). Do contrário, a execução continua no nodo 2 (ramo f) pois este é o destino implícito do desvio no nodo 1. Em contrapartida, os nodos 2 e 4 possuem instruções de desvio incondicional, representados pelo arco único de saída. O nodo 3 apresenta um caso especial, pois apesar de não possuir nenhuma instrução de desvio, ele também é considerado um nodo independente, por constituir um bloco básico da mesma forma que os demais nodos. Blocos básicos são detalhados no Capítulo 4.

Na arquitetura MIPS, existem instruções que podem gerar *exceções* que precisam ser tratadas imediatamente para que a execução do programa possa continuar [41]. Por exemplo, a instrução `lw $t0, 0($fp)` pode gerar uma exceção caso o endereço de memória indicado no registrador `$fp` seja inválido, por exemplo.

Em comparação com a desmontagem linear, a desmontagem recursiva apresenta menor eficiência, tanto pela forma de percorrer as instruções quanto pela necessidade de decodificar, pelo menos parcialmente, cada uma das instruções. Apesar disso, diversos trabalhos adotam esta técnica ([9; 12; 13; 16; 17; 18; 23; 25; 27; 31; 43]), especialmente pela sua maior precisão de desmontagem.

Uma característica negativa e bastante comum é o aparecimento de *sombras* durante a desmontagem, que são regiões do fluxo de instruções que não podem ser determinados estaticamente. Um exemplo típico desta situação é apresentado na Figura 3.3, onde grande parte das instruções não foi desmontada pois nenhuma referência estática que levasse ao bloco sombreado fora encontrada. No exemplo da Figura, a partir da instrução `jalr $v1` um desmontador recursivo tradicional não

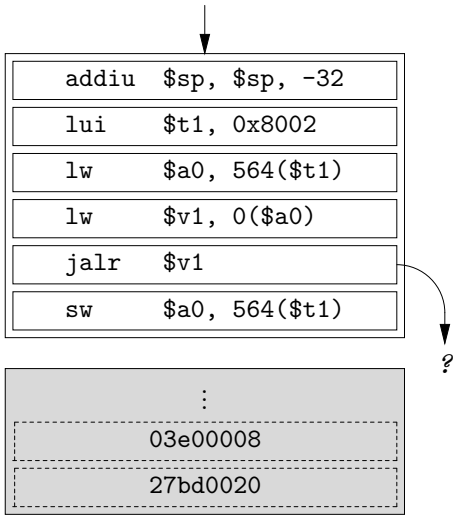

Figura 3.3: Exemplo de código apenas parcialmente desmontável estaticamente, devido à instrução de desvio `jalr` cujo destino não pode ser facilmente determinado, fazendo com que o restante do código fique *sombreado*.

tem condições de continuar, pois não é capaz de determinar o destino para o desvio incondicional encontrado, visto que o valor do registrador `$v1` seria dificilmente calculado estaticamente.

Esta situação é contornada por desmontadores recursivos em tempo de execução [16; 17; 27], conhecidos como *desmontadores dinâmicos*, que desmontam o código ao mesmo tempo em que permitem sua execução em um ambiente controlado, tendo então acesso às informações presentes apenas em tempo de execução. No exemplo da Figura 3.3, um desmontador dinâmico teria acesso ao conteúdo do registrador `$v1`, podendo prosseguir com a desmontagem. Contudo, apenas os trechos efetivamente executados são desmontados, sendo que os demais permanecem "sombreados".

Alguns trabalhos optam pela *execução simbólica*, isto é, modelam o comportamento do processador executando simbolicamente o código [13; 14; 15]. Assim, conseguem cobrir uma região maior do fluxo de instruções quando capazes de determinar simbolicamente destinos que dependeriam da execução do código, mas sem a complexidade intrínseca dos ambientes virtuais de execução.

A Figura 3.4 ilustra um exemplo simples de um trecho de código que não pode ser desmontado pela técnica recursiva convencional, mas sim com análise simbólica. Ao conseguir determinar, sem ambigüidade, o conteúdo do registrador `$v0`, o desmontador poderá continuar a desmontagem no endereço `0x0804e304`, como se a referência fosse estática.

3.3 Desmontagem Especulativa

A desmontagem especulativa não é propriamente uma técnica de desmontagem, mas sim extensões do funcionamento de outras técnicas, aumentando sua capacidade de desmontagem [18; 25; 43]. Alguns autores adotam o termo *desmontagem híbrida* para indicar a união da desmontagem especulativa com alguma outra técnica de desmontagem, em geral a recursiva [10].

Seu funcionamento está comumente associado a heurísticas provenientes de in-

	Instrução	Interpretação
`addiu`	`$v0, $zr, 0xe304`	`$v0` $\leftarrow$ 0+`0xe304`
`lui`	`$v0, 0x0804`	`$v0` $\leftarrow$ `$v0`+(`0x0804` $\times 2^{16}$)
`jr`	`$v0`	`$v0 = 0x0804e304`

Figura 3.4: Emprego da análise simbólica como forma de determinar estaticamente um destino. Ao simular o processamento que seria realizado durante a execução real do código, o desmontador computa o valor dos registradores, como apresentado na coluna *Interpretação*, sendo capaz de determinar o destino de um desvio referente ao conteúdo de um registrador.

formações como sistema operacional, arquitetura-alvo, formato do arquivo-objeto, linguagem de alto nível, compilador, e outras informações que porventura ofereçam detalhes sobre o código gerado [11]. Baseando-se nestas e outras informações, o desmontador avalia a probabilidade de um determinado trecho de dados constituir um fluxo de instruções, desmontando-o se a estimativa for favorável.

Esta técnica é útil para situações onde um binário tenha passado por um estágio de *ofuscamento*, uma técnica que tem por objetivo dificultar ou mesmo impossibilitar a desmontagem ou outros tipos de análises relacionadas à engenharia reversa [8; 13].

Alguns trabalhos apresentam resultados obtidos com a utilização da desmontagem especulativa em conjunto com a linear ou recursiva. Em especial, [10; 18] propõem a utilização da desmontagem especulativa também como forma de verificar a precisão da desmontagem realizada.

Devido à maior complexidade inerente à técnica, a eficiência da desmontagem é significativamente reduzida, tendo em vista que os resultados parciais devem ser freqüentemente comparados entre si para identificar possíveis inconsistências. Ainda, heurísticas adicionais permitem ao desmontador retornar a um estado anterior, descartando trechos desmontados considerados inconsistentes, e prosseguir a desmontagem do restante do binário, o que reduz ainda mais a eficiência de desmontagem.

3.4 Decodificação de Instruções

O estágio de decodificação de instruções é realizado conjuntamente com o de desmontagem. Ao determinar o formato da instrução, permite ao desmontador reconstruir o fluxo de controle do código sendo desmontado, os possíveis destinos ao encontrar uma instrução de desvio, e em alguns casos determinar o endereço da próxima instrução (como no da arquitetura Intel `80x86`).

A arquitetura MIPS possui 3 formatos básicos de instruções, ilustrados na Figura 3.5, sendo que todas as instruções possuem o mesmo tamanho (32 bits, ou 4 bytes) [39; 41; 44]. Na Figura, a significância de bits é apresentada acima das instruções (sendo 31 o bit mais significativo), e abaixo de cada campo o seu tamanho em bits O primeiro formato, *R*, é apropriado para operações que envolvam apenas registradores. O segundo formato, *I*, é usado em operações envolvendo até dois registradores, e um valor imediato de 16 bits. Finalmente, o terceiro formato, *J*, é usado para desvios maiores dos que os permitidos pelo formato *I*.

Dependendo da técnica de desmontagem adotada e do formato da instrução, a

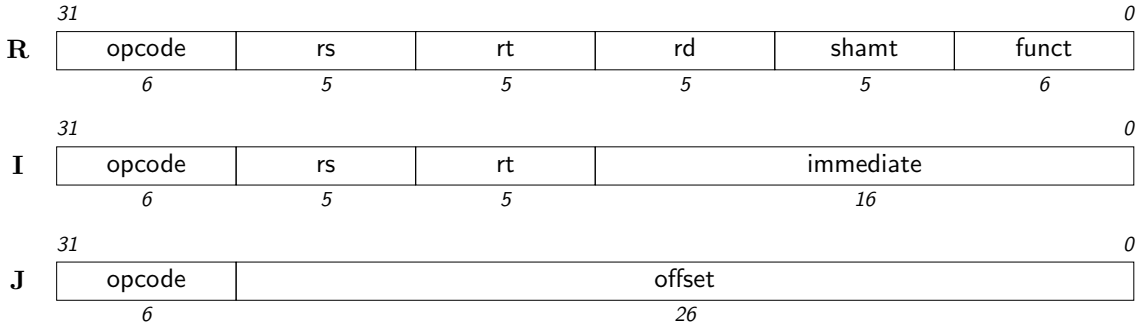

Figura 3.5: Formatos básicos de instruções da arquitetura MIPS. Os números abaixo de cada campo indicam seu tamanho em bits, enquanto que os números nos cantos superiores indicam a orientação dos bits, sendo 31 o bit mais significativo.

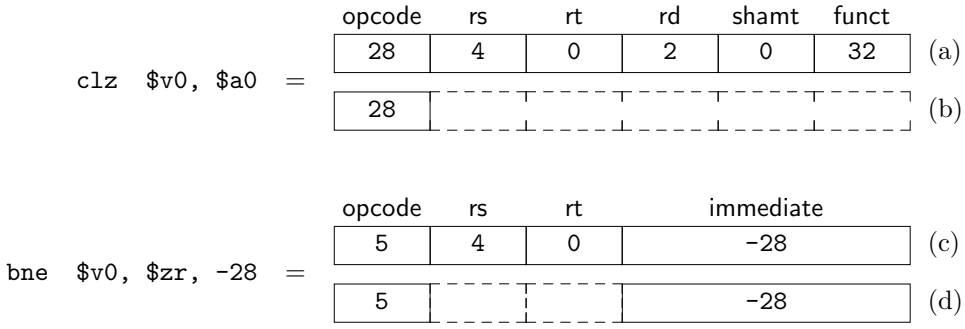

Figura 3.6: A desmontagem recursiva pode empregar a decodificação parcial de instruções, a fim de reduzir o processamento necessário para a desmontagem. A decodificação completa das instruções no lado esquerdo da Figura é apresentado em (a) e (c). Contudo, não é necessário decodificá-las completamente durante a desmontagem. Devido aos grupos de instruções da arquitetura MIPS, a primeira instrução pode ser suficientemente decodificada analisando-se apenas o campo *opcode* (b). Já a segunda instrução requer a análise de um campo adicional, *immediate* (d), para que seja prosseguir com a desmontagem.

decodificação de instruções dependerá da avaliação de campos adicionais. No contexto deste trabalho, a maioria das instruções MIPS não precisa ser completamente avaliada. A Figura 3.6 exemplifica esta situação. Em função dos diferentes formatos de instrução, a decodificação completa da primeira instrução no lado esquerdo da Figura requer a avaliação de seis campos distintos, como ilustrado em (a), enquanto que a segunda requer a avaliação de quatro campos. Contudo, a maior parte destas informações não são relevantes para o contexto da desmontagem. No primeiro caso, a instrução é suficientemente decodificada avaliando-se apenas o campo *opcode*, como ilustrado em (b). Já o segundo caso requer a avaliação de um campo adicional, *immediate* (ilustrado em (d)), para que o desmontador tenha condições de determinar os dois pontos onde a desmontagem prosseguirá.

Na arquitetura MIPS é comum o agrupamento de instruções, isto é, instruções que possuem um mesmo *opcode* mas diferenciam-se pelo campo *funct* (ilustrado na Figura 3.5), ou algum dos campos reservados para especificar os registradores envolvidos na operação, variando de acordo com as instruções. A Tabela 3.1 apresenta um grupo de instruções, sendo que todas possuem o mesmo *opcode*. Dependendo da

Tabela 3.1: Para agilizar a desmontagem, um desmontador pode tirar proveito dos grupos de instruções presentes no conjunto de instruções MIPS. Esta tabela apresenta um grupo de instruções cujo *opcode* = 28, distinguíveis pelo conteúdo do campo *funct*.

funct	Mnemônico	Descrição
0	`madd`	Multiplicação de duas palavras presentes nos registradores `rs` e `rt`, somando o resultado ao conteúdo dos registradores especiais `hi` e `lo`.
1	`maddu`	Análoga à instrução `madd`, mas assume que as palavras nos registradores `rs` e `rt` sejam sem sinal.
2	`mul`	Multiplicação em 64 bits de duas palavras presentes nos registradores `rs` e `rt`, armazenando a palavra menos significativa no registrador `rd`.
4	`msub`	Análoga à instrução `madd`, mas subtrai ao invés de somar ao conteúdo dos registradores especiais `hi` e `lo`.
5	`msubu`	Análoga à instrução `msub`, mas assume que as palavras sejam sem sinal.
32	`clz`	Conta o número de bits com valor lógico igual a zero, iniciando no bit mais significativo da palavra do registrador `rs`, armazenando o resultado no registrador `rd`.
33	`clo`	Análoga à instrução `clz`, mas conta o número de bits com valor lógico igual a um.

análise sendo realizada, um desmontador pode decidir por não avaliar totalmente uma instrução ao identificar que o agrupamento referenciado no campo *opcode* não contém nenhuma instrução relevante, como é o caso da instrução `clz`[2] ilustrada na Figura 3.6 em (b) e (c).

[2]A instrução `clz` é o mnemônico para *count leading zeroes.*

4 FORMATO INTERNO DE REPRESENTAÇÃO

Conforme o processo de desmontagem decodifica as instruções, é necessário armazená-las de forma a preservar a estrutura subjacente do código. Para atender este requisito, empregou-se dois tipos de grafos, sendo um para modelar a estrutura de controle do código, e outro para modelar o fluxo de dados entre as instruções. Estes grafos constituem o formato interno de representação, podendo armazenar diretamente as instruções. Contudo, um elemento mais eficiente se faz necessário—os blocos básicos—, para evitar que os grafos tenham desnecessariamente uma quantidade grande de elementos.

Este Capítulo está organizado como segue. A Seção 4.1 apresenta o grafo de fluxo de controle, seguida pela Seção 4.2 que apresenta o grafo de fluxo de dados. A Seção 4.3 encerra o Capítulo apresentando formalmente o conceito de blocos básicos.

4.1 Grafo de Fluxo de Controle

Para que a análise estática de código executável seja possível, um analisador deve primeiramente determinar o *grafo de fluxo de controle* (CFG, *Control Flow Graph* em Inglês), usado durante a geração de código pelo compilador.

O CFG é gerado conforme o analisador processa as instruções, incluindo novos nodos e vértices. A Figura 4.1 ilustra em (a) um exemplo em linguagem C implementado em linguagem de montagem MIPS em (b). O resultado da análise de fluxo das instruções em (b) é o CFG apresentado em (c), onde cada nodo representa uma instrução de (b), e as setas representam o fluxo de controle.

Como o CFG representa a ordem de execução das instruções, é possível computar um grafo equivalente ao construído pelo compilador. Além disso, ao computar o CFG também é possível determinar os *blocos básicos* (detalhados na Seção 4.3) do código, elementos fundamentais para análises subseqüentes.

4.2 Grafo de Fluxo de Dados

De forma análoga ao grafo de fluxo de controle apresentado na Seção anterior, o *grafo de fluxo de dados* (DFG, *Data Flow Graph* em Inglês) modela as interdependências entre instruções em função de seus operandos. Devido à maior complexidade quando comparado ao CFG, é comum que um DFG seja computado apenas dentro dos blocos básicos que o analisador considerar relevantes [21].

A Figura 4.2 ilustra um exemplo simples de fluxo de dados, contrastando-o com o fluxo de controle. A seqüência de instruções MIPS em (a) resulta no CFG ilustrado

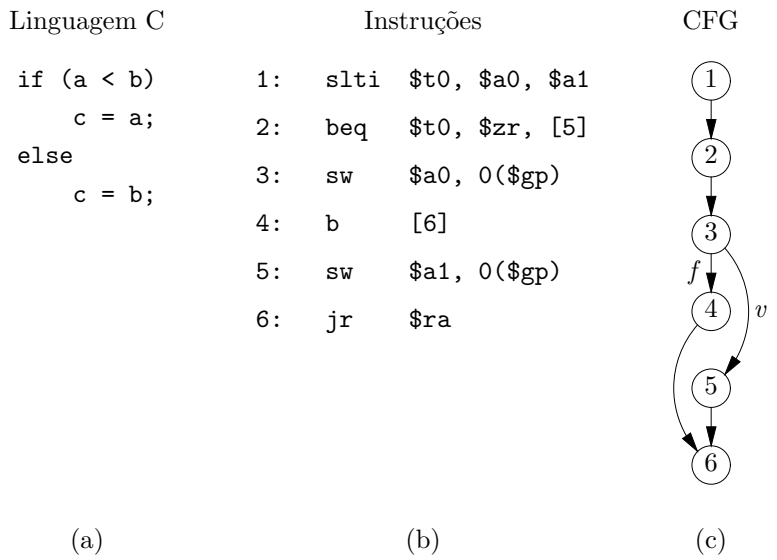

Figura 4.1: Grafo de fluxo de controle (CFG). O código em linguagem C apresentado em (a) pode ser implementado pelas instruções MIPS em (b), e estas representadas pelo CFG em (c). O vértice f partindo do nodo 3 indica que, caso o desvio não seja tomado, o próximo nodo será o 4. Do contrário, o próximo nodo será o 5, indicado pelo vértice v.

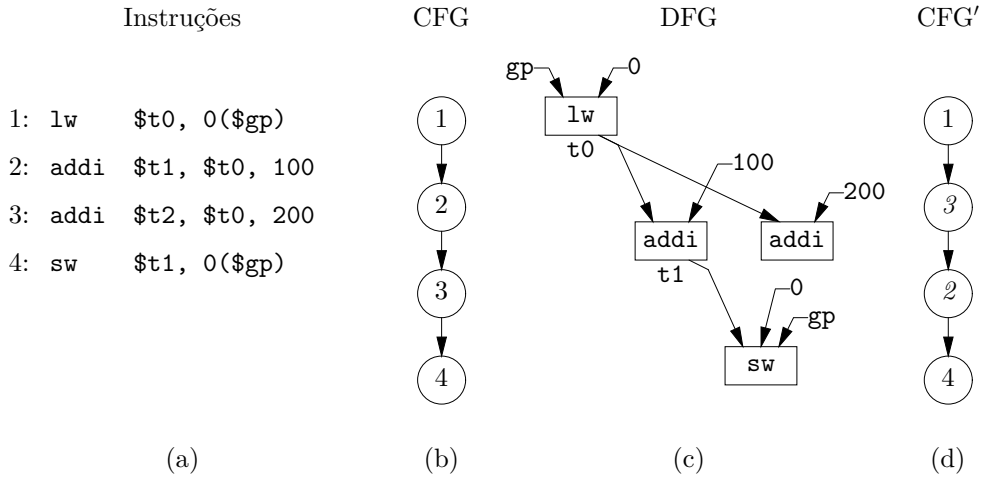

Figura 4.2: Grafo de fluxo de dados (DFG). As instruções MIPS em (a) geram o CFG ilustrado em (b). A análise de fluxo de dados sobre estas instruções gera o DFG (c), sendo que a partir dele pode-se identificar oportunidades para alterar a ordem de instruções sem comprometer a semântica do código. O CFG (d) foi gerado modificando-se as instruções 2 e 3 do CFG original, por não apresentarem interdependência.

em (b). Enquanto que o CFG indica que todas as instruções são executadas seqüencialmente, o DFG (c) apresenta dois possíveis fluxos que respeitam as dependências de dados. Pelo DFG, ambas as instruções `addi`[2] não apresentam interdependência de dados e, portanto, podem ter sua ordem de execução alterada sem comprometer a semântica desta computação, permitindo a geração de um segundo CFG, como ilustrado em (d). Nele as instruções 2 e 3 têm sua ordem invertida em comparação com o CFG inicial (b). Além disso, pelo DFG verifica-se que as instruções 1 e 4 não podem ter sua ordem alterada.

4.3 Blocos Básicos

As instruções que compõem um código executável podem ser agrupadas logicamente criando a noção de *blocos básicos*. Para que um determinado conjunto de instruções seja considerado um bloco básico, ele deve (*i*) ter um único ponto de entrada e outro de saída; (*ii*) compreender apenas instruções adjacentes; e (*iii*) constituir um bloco básico disjunto dos demais.

Ponto único de entrada e de saída. As instruções $x_1, \ldots, x_n$ farão parte de um mesmo bloco básico B se, e somente se, para cada instrução x_i com $1 \leq i \leq n$, x_i dominar as instruções $x_{i+1}, \ldots, x_n$. Em outras palavras, sempre que a instrução a instrução x_i for executada, as instruções subseqüentes até a x_n também o serão.

É importante notar que mesmo em um bloco cuja última instrução seja um desvio condicional (portanto tendo dois destinos possíveis), este será seu ponto único de saída. Desta forma, um conjunto contendo duas ou mais instruções de desvio não pode ser considerado um bloco básico, por violar esta premissa. A arquitetura MIPS, bem como outras arquiteturas RISC, adota *delay slots*, isto é, instruções após uma instrução de desvio que são executadas antes da instrução de desvio em si, estendendo, portanto, o ponto de saída do bloco básico de forma que compreenda as instruções presentes no *delay slot*. No caso da arquitetura MIPS, o *delay slot* é composto por uma instrução [41; 39].

Instruções adjacentes. As instruções $x_1, \ldots, x_n$ farão parte de um mesmo bloco básico B contendo n instruções se, e somente se, a instrução x_{i+1} é sucessora imediata da instrução x_i, para $1 \leq i < n$. Todas as instruções da arquitetura MIPS possuem 4 bytes (32 bits) [39; 44], portanto, uma instrução x_{i+1} no endereço M_b é sucessora imediata da instrução x_i no endereço M_a se, e somente se:

$$\frac{M_b - M_a}{4} = 1$$

Disjunção dos blocos básicos. Cada instrução x do conjunto de N instruções de um código executável só pode estar contida em apenas um bloco básico. Considerando o conjunto de n blocos básicos $B = B_1, \ldots, B_n$ de um código executável, eles são disjuntos se, e somente se, $B_i \cap B_j = \emptyset$ para $1 \leq i \leq n$ e $1 \leq j \leq n$, sendo $i \neq j$.

[2] `addi` é o mnemônico para *add immediate word* em Inglês, instruindo o processador a somar um valor imediato ao registrado destino especificado na instrução.

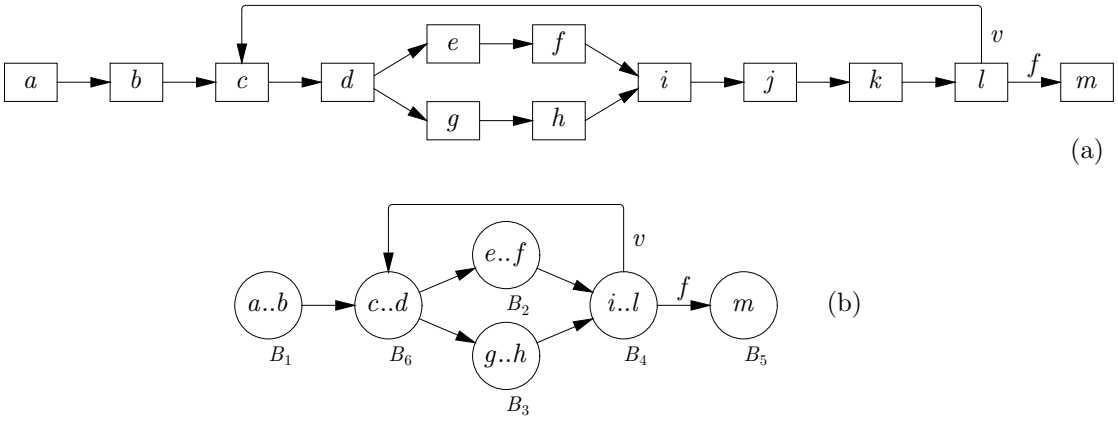

Figura 4.3: Formação de blocos básicos. Os nodos do grafo (a) representam instruções, criando a noção de blocos básicos quando agrupados, como ilustrado em (b). Além de ser equivalente ao grafo de instruções, o grafo de blocos básicos (b) possui significativamente menos nodos que o grafo (a).

A Figura 4.3 ilustra simbolicamente a formação de blocos básicos a partir de um conjunto de instruções que implementam o trecho de código na linguagem C da Figura 4.4 (a). Em um primeiro momento, um analisador parte do conjunto de instruções ilustrado em (a), determinando os blocos básicos $B_1 = \{a, b, c, d\}$, $B_2 = \{e, f\}$, $B_3 = \{g, h\}$, $B_4 = \{i, j, k, l\}$, e $B_5 = \{m\}$. Contudo, a instrução l confere ao bloco B_1 um segundo ponto de entrada, violando uma das premissas fundamentais. O analisador deve, portanto, dividir o bloco B_1 nos blocos $B_1 = \{a, b\}$ e $B_6 = \{c, d\}$, como ilustrado em (b). O processo de divisão de blocos básicos é detalhado na Seção 5.3. Ao final do processo, o analisador obtém o CFG de blocos básicos (b), que é equivalente ao CFG original de instruções (a).

Espera-se que o esforço computacional investido na determinação do CFG de blocos básicos seja compensado pela menor quantidade de recursos necessários para a manipulação do mesmo nas etapas subseqüentes. O exemplo ilustrado na Figura 4.3, embora simples, apresenta uma redução de 54% na quantidade de nodos quando se compara os CFGs apresentados em (a) e (b), respectivamente com 13 e 6 nodos, o que é bastante significativo considerando o tamanho reduzido do exemplo.

A Figura 4.4 apresenta a situação real anteriormente representada de forma simbólica na Figura 4.3. O trecho de código em linguagem C é apresentado em (a), sendo que (b) ilustra uma possível implementação para a arquitetura MIPS. Ainda, o resultado de uma desmontagem que empregue a técnica recursiva gerará os blocos básicos como ressaltados em (b). Estes, por sua vez, podem ser representados pelo CFG em (c), que é equivalente ao grafo apresentado na Figura 4.3 (b), anteriormente.

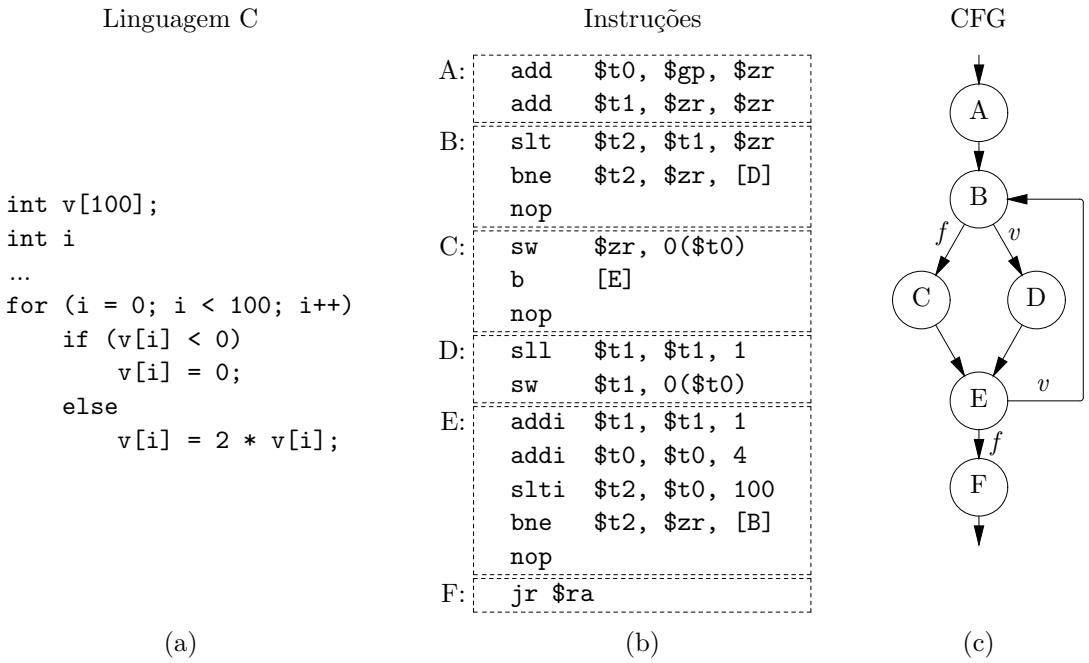

Figura 4.4: Exemplo real de formação de blocos básicos. O código fonte na linguagem C em (a) pode ser implementado pelo código em linguagem de montagem MIPS (b) e representado pelo CFG (c). Um desmontador recursivo que analise o código em (b) será capaz de recuperar os limites dos blocos básicos originais, gerando o mesmo grafo (c) a partir da entrada (b), sem conhecer o código fonte original (a).

5 DESMONTAGEM RECURSIVA ESTENDIDA

A técnica de desmontagem recursiva tradicional apresenta algumas limitações indesejáveis, que podem ser contornadas com o emprego da análise simbólica e também da desmontagem especulativa, ambas detalhadas no Capítulo 3. Contudo, estas alternativas aumentam consideravelmente a complexidade e a quantidade de recursos necessários.

Este Capítulo está organizado da seguinte forma. A extensão proposta à técnica recursiva tradicional é apresentada na Seção 5.1, seguida pela abstração adotada para mapear os blocos básicos em uma representação de memória virtual, detalhada na Seção 5.2. O processo de divisão de blocos básicos, apresentado na Seção 5.3, encerra o Capítulo.

5.1 Extensão à Técnica Recursiva Tradicional

Enquanto que desmontadores recursivos tradicionais iniciam com um conjunto vazio de informações e vão agregando os resultados parciais da desmontagem de forma incremental, a técnica proposta considera inicialmente que cada segmento executável de um código é um grande bloco básico, dividindo-o sucessivamente em blocos básicos menores de acordo com o fluxo de desmontagem.

A Figura 5.1 ilustra este processo de forma resumida. Antes de iniciar a desmontagem, todos os segmentos executáveis do arquivo-objeto ELF são carregados, sendo que cada um deles é considerado como um bloco básico. O exemplo (a) da Figura constitui apenas um destes blocos, sendo representado pelo CFG em (b). Em seguida a desmontagem é iniciada no ponto de entrada do código, indicado no cabeçalho principal do arquivo ELF, prosseguindo de exatamente como a técnica recursiva tradicional. Assim que um limite interno ao bloco básico é identificado, o bloco básico é *dividido* em outros blocos, como ilustrado em (c). A desmontagem prossegue recursivamente em cada novo bloco básico ainda não processado, representados pelos nodos escuros na Figura, dividindo múltiplas vezes o bloco inicial, como ilustrado em (d) e resultando no CFG″.

5.2 Mapeamento Virtual

O mapeamento virtual de memória é uma estrutura de dados que representa o endereçamento de memória do código sendo analisado. Este mapa é basicamente uma lista duplamente encadeada, onde cada elemento é, na verdade, um ponteiro para nodos do CFG, que representam blocos básicos. Portanto, o mapa virtual de

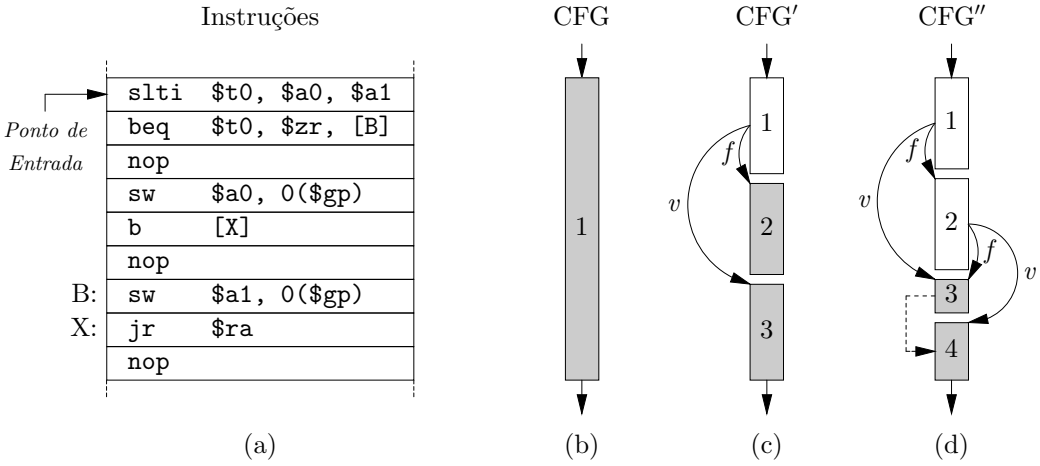

Figura 5.1: Visão geral da desmontagem recursiva estendida. Em um primeiro momento, o desmontador assume que as instruções MIPS em (a) constituem um único bloco básico ainda não analisado, representado pelo nodo escurecido no CFG (b). Conforme analisa o bloco, o desmontador atualiza as bordas do bloco 1 e determina outros 2 blocos (nodos 2 e 3), como ilustrado em (c). Ao final da análise deste trecho de instruções, o desmontador gera o CFG″ ilustrado em (d). O vértice que liga os blocos 3 e 4, representado pela linha tracejada em (d), é um vértice especial usado para concatenar blocos básicos, não estando associado à semântica do programa, servindo apenas para auxiliar os processos de desmontagem, manipulação e remontagem.

memória é uma lista duplamente encadeada dos blocos básicos que representam o código.

A lista deve ser mantida ordenada o tempo todo, e para isso o algoritmo de inserção ordenada (*insertion sort*, em Inglês) é usado. Os blocos são ordenados em função de sua posição em memória e tamanho. Durante a inserção de um elemento na estrutura, primeiramente a posição correta na lista é localizada, em função do endereço virtual base do bloco básico a ser inserido. Por questões de consistência, verifica-se se o bloco não se sobrepõe a nenhum outro bloco já existente na lista, sendo que neste caso a execução da ferramenta deve ser interrompida.

Esta estrutura é bastante importante para garantir a eficiência da ferramenta, pois permite que qualquer bloco básico seja localizado rápida e facilmente, sem a necessidade de se percorrer o grafo. Embora o tempo seja proporcional ao tamanho da lista ($O(n)$), esta estrutura também pode ser implementada como uma estrutura do tipo árvore para reduzir o tempo esperado do pior e médio casos. Devido ao tempo disponível, optou-se por uma estrutura de implementação mais simples.

5.3 Divisão de Blocos Básicos

Conforme os blocos básicos são processados, o desmontador divide os blocos analisados em blocos menores, aumentando gradativamente a quantidade de nodos do CFG. O tipo de divisão em si dependerá do contexto em que o desmontador se encontra, e é resumido no Algoritmo 5.1.

Linhas 1–2 O desmontador decodifica parcialmente as n instruções do bloco básico B a partir do início, iterando sobre elas até encontrar uma que altere o fluxo

Algoritmo 5.1 Divisão de blocos básicos

Entrada: Bloco básico B contendo n instruções, sendo $n \geq 3$

```
1:  for i = 1 to n do
2:    if desvio(B_i) then
3:      B' = {B_1, ..., B_{i+1}}
4:      B'' = {B_{i+2}, ..., B_n}
5:      if desvio_condicional(B_i) then
6:        e_v = destino(B_i)
7:        D = bloco_contendo(e_v)
8:        if |D| ≥ 2 then
9:          j = posição(e_v, D)
10:         D' = {D_1, ..., D_{j-1}}
11:         D'' = {D_j, ..., D_n}
12:         D'_suc = D''
```

de controle, ou atinja o final do bloco.

Linhas 3–4 No caso de encontrar um desvio (instrução x_i), o bloco B é dividido gerando os blocos B' e B'' que conterão, respectivamente, as instruções x_1 a x_{i+1} e x_{i+2} a x_n. É importante lembrar que a instrução colocada no *delay slot*, B_{i+1}, permanece no bloco original, já que ela é executada independentemente do tipo de desvio [42].

Linhas 5–7 O desmontador verifica se o desvio é condicional, computando o destino caso a condição seja verdadeira (endereço de memória e_v) e recuperando uma referência ao bloco básico D que contém este endereço, se necessário.

Linhas 8–11 Neste momento, o bloco destino D é avaliado para identificar se o endereço e_v referencia sua primeira instrução ou se o bloco D deve ser dividido. O primeiro caso é trivial, pois ou e_v referencia a instrução D_1 ou o bloco D contém apenas uma instrução e portanto não pode ser dividido. Neste último caso, o desmontador apenas referencia o bloco D. Se a divisão for necessária, ele primeiro computa a posição da instrução destino d_j, e divide o bloco D' nos blocos D'' na posição da instrução d_j, sendo que os novos blocos conterão, respectivamente, as instruções d_1 a d_{j-1} e d_j a d_m.

Linha 12 Um vértice de concatenação é inserido de D' para D'', indicando que o nodo D'' é o *sucessor implícito* de D'. No contexto deste trabalho, um bloco B é *sucessor implícito* de um bloco A se (i) o bloco A não possui instruções que alterem o fluxo de controle, e (ii) caso a instrução A_{n+1} seja B_i, ou seja, a próxima instrução à última instrução do bloco A é a primeira instrução do bloco B, sendo B_i sucessora imediata de A_n.

Este algoritmo caminha recursivamente pelos novos ramos descobertos em função da análise de um determinado bloco, dividindo-os. O comportamento das divisões pode ser melhor entendido analisando-se separadamente em função da origem e dos destinos envolvidos.

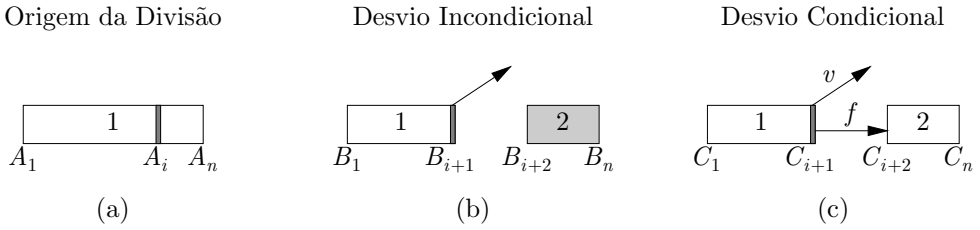

Figura 5.2: Divisão do bloco de origem. O bloco (a) que contém a instrução de desvio na posição A_i, ao ser dividido, pode gerar duas situações, dependendo do tipo de desvio. A configuração (b) é adotada caso o desvio seja *incondicional*, onde o bloco 1 não apresenta ligação qualquer com o bloco 2. Já no caso de um desvio *condicional*, o bloco 2 é o destino implícito do bloco 1 caso a condição não seja satisfeita (ramo f). Em ambos os casos, o *delay slot* (posições B_{i+i} e C_{i+1}) permanece no bloco de origem.

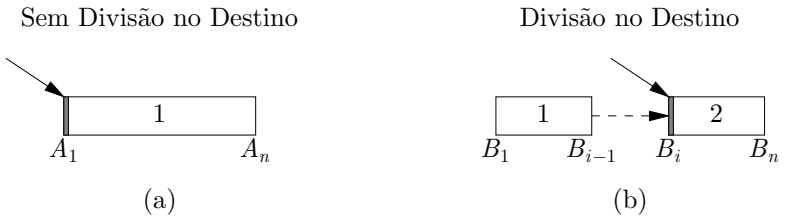

Figura 5.3: Divisão do bloco de destino. Quando a divisão ocorre no bloco de destino, uma das configurações ilustradas será adotada. No caso (a), considerado trivial, o bloco não necessita de divisão, pois a referência aponta para a primeira instrução do bloco. Já em (b), o bloco é dividido nos blocos 1 e 2 na instrução B_i, sendo que um vértice de concatenação é inserido ligando o bloco 1 ao 2.

A Figura 5.2 ilustra as duas possíveis modificações resultantes da divisão, considerando apenas ponto de origem, isto é, o nodo que será o antecessor dos novos nodos. Ao encontrar uma instrução de desvio no bloco 1 em (a), ele será dividido no ponto A_i (mantendo a instrução do *delay slot* no bloco de origem, como mencionado anteriormente), podendo resultar nas configurações ilustradas em (b) e (c). A primeira (b), ocorre quando a instrução de desvio é *incondicional*, sendo que o bloco 2 passa a ser um bloco independente dos demais, por não possuir uma referência até este ponto da desmontagem. Caso o desvio em (a) seja *condicional*, a configuração (c) é utilizada, sendo que o bloco 2 é considerado o destino implícito do nodo 1 caso a condição de desvio seja falsa (ramo f).

Diferentemente da técnica recursiva tradicional, os blocos básicos que não são referenciados por outros blocos (isto é, blocos sombreados, como ilustrado na Figura 5.2 (b)) ainda o são pelo mapeamento virtual de memória. Esta característica se mostrará especialmente importante durante a remontagem, que é realizada com base no mapeamento virtual de memória e não apenas pelo CFG. As Seções 6.3 e 7.3 apresentam detalhes do processo de remontagem.

Em contrapartida, a Figura 5.3 ilustra os possíveis resultados da divisão sob o ponto de vista do destino de uma instrução de desvio. No caso de não ser necessário dividir um bloco, a configuração (a) é adotada. Isto ocorre quando o destino de um desvio referencia a primeira instrução de um bloco básico ou ele não pode ser

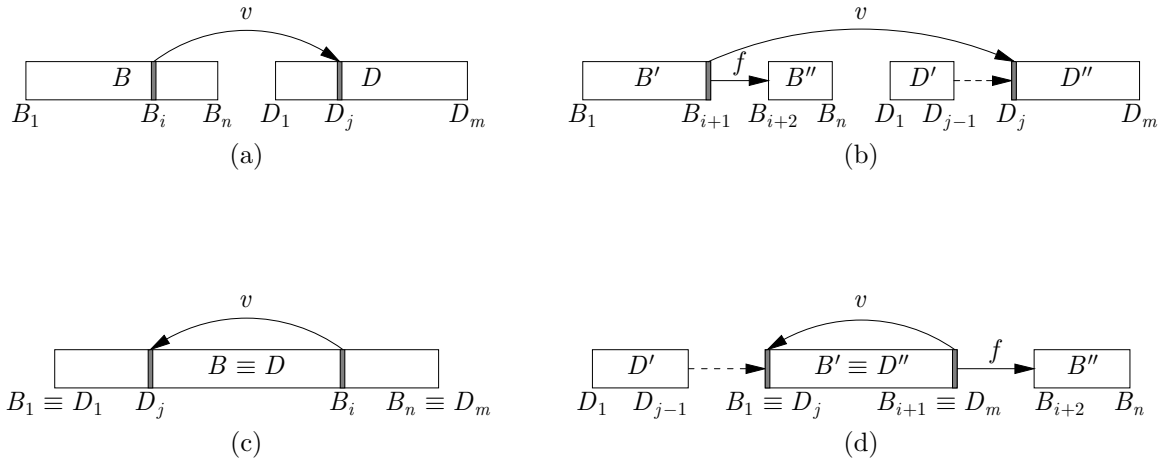

Figura 5.4: Exemplos de divisão de blocos básicos. Em (a) um desvio condicional é encontrado em B_i, resultando na divisão ilustrada em (b). O bloco B é dividido nos blocos B' e B'', assim como D nos blocos D' e D''. O destino implícito do desvio no bloco B' é o bloco B''. Devido à divisão, o bloco D' é ligado ao D'' com um vértice de concatenação. O caso em (c) apresenta um bloco contendo uma instrução de desvio que referencia o próprio bloco. Neste caso, há uma equivalência entre o bloco de origem e de destino, embora a divisão proceda da mesma maneira.

dividido. Já a configuração ilustrada em (b) ocorre quando o bloco destino precisa ser dividido. Neste caso, o ponto de divisão é calculado e o bloco é dividido. Além disso, um vértice de concatenação é inserido ligando o nodo 1 ao nodo 2. Assim que o bloco 1 for analisado, se for constatado que ele possui uma instrução de desvio, o vértice será removido.

A Figura 5.4 apresenta um resumo das situações que podem ocorrer durante a divisão de blocos básicos. Na parte superior da Figura, em (a), uma instrução de desvio condicional é encontrada na posição B_i do bloco B, cujo destino é a instrução D_j do bloco D. O resultado desta divisão é apresentado em (b). O bloco B é dividido nos blocos B', contendo as instruções B_1 a B_{i+1} (incluindo o *delay slot*), e B'', que contém as instruções B_{i+2} a B_n. O destino implícito de B', representado pelo vértice f, que contém a instrução de desvio, é o bloco B''. Já o bloco D é dividido nos blocos D', contendo as instruções D_1 a D_{j-1}, e D'', contendo as instruções D_j a D_m. No caso do bloco D, em função de ter ocorrido uma divisão, um vértice de concatenação é inserido ligando o bloco D' ao D''.

A parte inferior da Figura 5.4 generaliza a situação anterior para o caso onde o bloco sendo analisado contém uma instrução de desvio que referencia ele próprio na posição B_i. Em (c) uma instrução de desvio condicional foi encontrada na posição B_i, como no caso anterior. Contudo, o destino D do desvio é o próprio bloco B, portanto $B \equiv D$. Diferentemente do caso anterior, o bloco $B \equiv D$ é dividido nos pontos B_i e D_j. O bloco D' conterá as instruções D_1 a D_{j-1}. Já o bloco D'' contém as instruções D_j a D_m, sendo que este intervalo compreende as instruções B_1 a B_{i+1}, equivalente ao bloco B'. Portanto, $B' \equiv D''$, $B_1 \equiv D_j$, e $B_{i+1} \equiv D_m$. O bloco B'' contém as instruções B_{i+2} a B_n. De forma semelhante à ilustrada em (b), o destino implícito do bloco $B' \equiv D''$ é B'', e um vértice de concatenação é inserido ligando o bloco D' ao bloco $B' \equiv D''$.

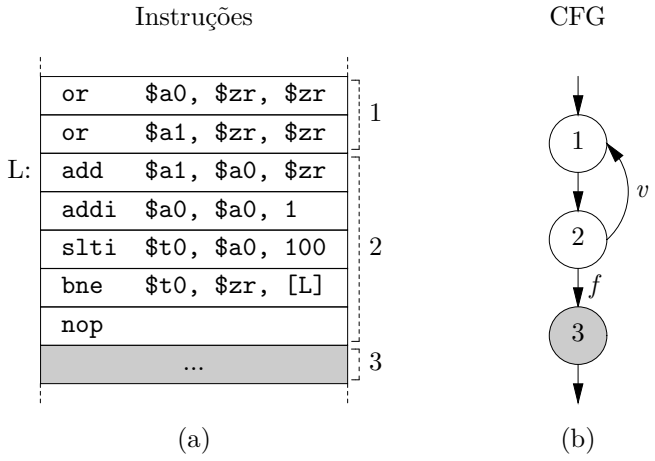

Figura 5.5: Exemplo de ciclo em um CFG. A seqüência de instruções MIPS em (a) apresenta um ciclo, como pode ser visto no CFG resultante em (b), e também no CFG de blocos básicos em (c).

Pela natureza da técnica de desmontagem recursiva, existe a possibilidade do desmontador processar indefinidamente o mesmo bloco ao se deparar com um ciclo no CFG. A Figura 5.5 ilustra em (a) um trecho de instruções MIPS que geram um ciclo em seu respectivo CFG, como ilustrado em (b). Se o desmontador não empregar uma forma de evitar a análise repetida de blocos, ele aplicará o algoritmo de divisão no bloco 1 constantemente, pois este possui um ramo apontando para seu início. Considerando que ciclos como o exemplificado na Figura 5.5 são freqüentemente usados para implementar construções de laços tais como `for`, `while` e `do...while` da linguagem C, um desmontador incapaz de lidar com ciclos no código será de pouca utilidade.

6 INFRA-ESTRUTURA PROPOSTA

Para reduzir a complexidade de manipulação de dados e para que as soluções propostas neste trabalho possam ser aplicadas a diferentes arquiteturas-alvo[1], a infra-estrutura aqui proposta é modular e extensível, composta por estágios, módulos, e controle de fluxo de trabalho. A Figura 6.1 ilustra esta infra-estrutura. Nela apresenta-se um coordenador de estágios responsável por determinar a seqüência de atividades aplicadas sobre os dados de entrada. Cada estágio é composto por um conjunto de módulos que exercem atividades específicas. Ao final de cada estágio, obtém-se dados intermediários que são passados como entrada para o próximo estágio, sendo que após o último estágio obtém-se os dados de saída.

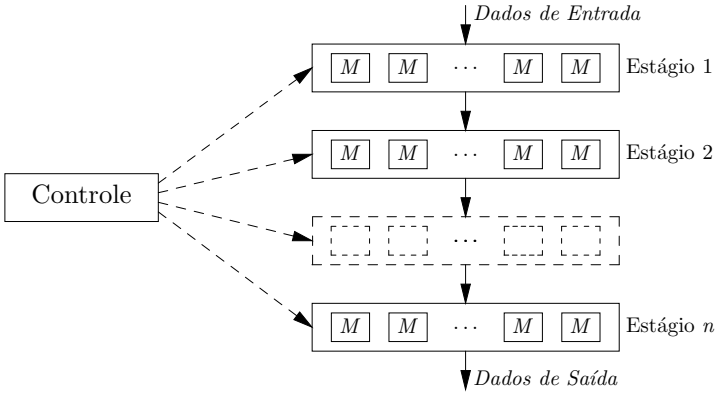

Figura 6.1: Abordagem modular e extensível adotada. A seqüência de atividades, representa pelas setas tracejadas, é determinada pela unidade de controle. Cada estágio é composto por módulos que exercem atividades específicas sobre os dados de entrada, resultando em dados de saída após o último estágio. As setas sólidas representam o fluxo de dados através de n estágios.

De uma forma bastante sucinta, os estágios podem ser divididos em *entrada de dados*, onde extrai-se código executável de um programa para um formato de representação intermediária; *manipulação de instruções*, onde o fluxo de instruções é modificado a fim de obter-se um comportamento diferente do original; e finalmente o estágio de *saída de dados*, no qual o formato intermediário, possivelmente modificado, é convertido em um novo binário.

[1]Como mencionado na Introdução (Capítulo 1) *arquitetura-alvo* é, no contexto deste trabalho, a combinação de formato de arquivo-objeto, sistema operacional, e processador para os quais um determinado código executável é gerado.

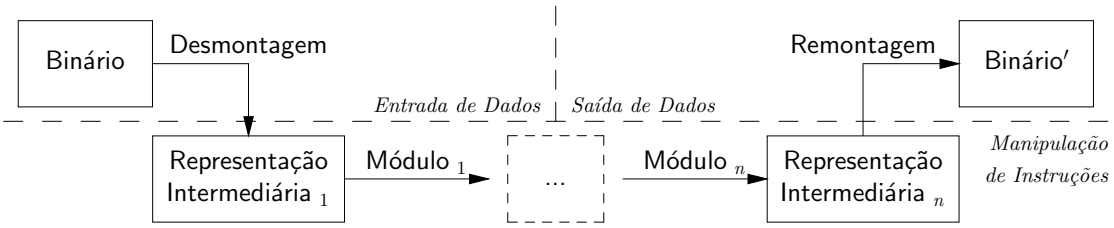

Figura 6.2: Visão geral da infra-estrutura. Um binário é fornecido como entrada, sendo inicialmente desmontado, gerando o primeiro estado da representação intermediária. Esta representação é modificada pela aplicação de n módulos de manipulação, resultando no estado n da representação intermediária. Um binário modificado é gerado a partir deste último estado da representação intermediária, pelo processo de remontagem.

A Figura 6.2 ilustra os três principais estágios da infra-estrutura—desmontagem, manipulação de instruções e remontagem—, apresentando uma visão geral do fluxo de trabalho da abordagem proposta. Inicialmente, um binário é desmontado no estágio de entrada de dados, gerando uma representação intermediária. Este primeiro estado da representação intermediária é modificado por n módulos no estágio de manipulação de instruções, resultando no estado final n da representação intermediária. Um novo binário é então remontado no estágio de saída de dados, a partir deste último estado da representação intermediária.

6.1 Entrada de Dados

Este estágio é responsável por converter código contido em um arquivo executável para uma representação interna manipulável, e pode ser subdividido em três componentes bem definidos: *identificador*, *decodificador* e *desmontador*.

6.1.1 Identificador

Em um primeiro momento, os dados de entrada precisam ser identificados a fim de determinar qual o módulo apropriados para realizar a decodificação. Existem basicamente três formas de se identificar o formato de dados de arquivos sem depender de fontes externas (como por exemplo meta-dados), que podem ser combinadas para tornar a identificação mais precisa [45].

A primeira delas baseia-se em características de seu nome, geralmente o sufixo ou extensão[2]. Devido à abordagem simplista, esta técnica é pouco confiável e propensa a erros, em especial por apoiar-se na boa fé do usuário. Além disso, ela é pouco efetiva em ambientes UNIX, já que uma parcela considerável dos arquivos não possuem sufixo, em especial os binários executáveis.

A segunda técnica de identificação considera uma *assinatura* do arquivo. Embora não exista uma especificação formal, a assinatura de um arquivo está geralmente associada à um *número mágico* (*magic number*, em Inglês) sem tamanho pré-definido, localizado em uma determinada posição no arquivo (própria do formato), tradicionalmente nos bytes iniciais do mesmo. O número mágico é um identificador geralmente escolhido pelo autor do formato que, apesar de numérico, geralmente possui

[2]O sufixo ou extensão de um arquivo cujo nome seja `aaa.bbb` é `bbb`, isto é, o conteúdo após a última ocorrência do caractere `.` (ponto). O sufixo de um arquivo `aaa.bbb.ccc` é, portanto, `ccc`.

um significado diferente do expresso pelo valor numérico ou bytes que o compõem. Por exemplo, o número mágico adotado em arquivos-objeto ELF é composto pelos números em base decimal 127, 69, 76, e 70, sendo que os três últimos representam os valores atribuídos aos caracteres `E`, `L`, e `F`, respectivamente, no conjunto ASCII[3] de caracteres.

Finalmente, a terceira técnica baseia-se em heurísticas a respeito do conteúdo do arquivo, que é inspecionado em busca de elementos-chave que levem à sua identificação. Uma ferramenta que adote esta técnica pode, por exemplo, procurar por construções específicas da linguagem C para identificar um arquivo textual que, na verdade, contenha um código fonte nesta linguagem. Porém, é possível que tal ferramenta faça uma identificação do tipo *melhor esforço*, isto é, com uma probabilidade associada, especialmente para formatos sem estrutura rígida.

6.1.2 Decodificador

Com o módulo de decodificação devidamente selecionado pelo identificador, os dados são lidos do arquivo e convertidos para a representação intermediária adotada. No caso de arquivos-objeto ELF, seus segmentos executáveis são carregados e inseridos no mapeamento virtual de memória, sendo que cada um deles constitui um grande bloco básico neste primeiro momento (como detalhado no Capítulo 5).

O decodificador deve ser capaz de, portanto, (*i*) interpretar as informações específicas do formato do arquivo-objeto, (*ii*) localizar segmentos executáveis, (*iii*) adequar os dados carregados para o formato apropriado, especificamente corrigindo a ordem de bytes dos mesmos, e finalmente (*iv*) popular as estruturas de dados intermediárias, para que o estágio de desmontagem possa ser realizado.

Após sua execução, o decodificador terá populado o mapeamento de memória virtual, retornando uma referência ao elemento onde se encontra o ponto de entrada do programa, isto é, onde a desmontagem deve ser iniciada.

6.1.3 Desmontador

Assim que o código executável do arquivo de entrada é extraído e disposto no mapeamento virtual de memória, o módulo de desmontagem definido pelo identificador é invocado para processar as instruções.

O desmontador é responsável por popular as estruturas de dados de maior nível de abstração, como o CFG e DFG (detalhados no Capítulo 4), empregando a técnica de desmontagem recursiva estendida (detalhada no Capítulo 5). Embora a técnica seja a mesma para as diversas arquiteturas, cada uma apresenta peculiaridades, como por exemplo a arquitetura MIPS, que possui os conjuntos de instruções MIPS32 e MIPS64.

Como entrada, o desmontador recebe uma referência para o nodo que contém o ponto de entrada do programa, iniciando a desmontagem a partir dele, sendo que após o processamento ele retorna uma referência ao bloco básico que representa o nodo raiz do CFG.

[3] *American Standard Code for Information Interchange*

6.2 Manipulação de Instruções

O estágio de manipulação de instruções é implementado por múltiplos módulos de manipulação, sendo que a ordem e quantidade de aplicações, bem como quais módulos serão aplicados, ficam a critério do usuário. O tipo de modificação depende exclusivamente da implementação do módulo, variando desde a aplicação de uma técnica de otimização até a tradução binária entre diferentes arquiteturas, por exemplo. O Capítulo 7 detalha um módulo de manipulação que implementa uma técnica de otimização, como forma de validação da solução proposta.

Cada módulo de manipulação tem como entrada uma referência ao nodo raiz do CFG, e a partir daí realiza o processamento e modificações de acordo com suas características. O CFG oferece apenas duas granularidades de abstração (blocos básicos e instruções de máquina, sendo esta última a menor), embora a existência de outras simplifique consideravelmente a implementação de módulos complexos. Por exemplo, uma ferramenta que ofereça a noção de *macro blocos*[4] reduz a complexidade de implementação de módulos para certos tipos de otimizações inter- ou intra-procedural, em especial módulos que implementem a fusão de blocos básicos, com o objetivo de reduzir a quantidade de instruções de desvios [21].

A saída dos módulos de manipulação é um CFG modificado, que é então passado como entrada para o próximo módulo, se houver, ou devolvido ao coordenador. Do ponto de vista de implementação, os módulos modificam diretamente o CFG indicado pelo coordenador, não havendo portanto, duplicação ou outras operações que demandariam recursos adicionais.

6.3 Saída de Dados

No contexto deste trabalho, o estágio de saída de dados representa a geração de um arquivo-objeto ELF válido a partir dos dados de entrada, possivelmente modificados. De forma análoga aos componentes de entrada de dados, o estágio de saída de dados também é organizado sob a forma de componentes, sendo um responsável pela consolidação da *imagem executável*[5] e outro pela criação do novo arquivo-objeto. Esta abordagem possibilita a tradução entre arquivos-objeto de diferentes formatos, embora este não seja o foco do trabalho proposto.

6.3.1 Consolidação

O módulo de consolidação tem por objetivo possibilitar a criação de uma nova imagem executável a partir da imagem original possivelmente modificada pelos módulos de manipulação. Sua principal atribuição é o *ajuste de endereços* referenciados pelas instruções de desvio, visto que a simples inclusão ou remoção de uma única instrução por algum módulo de manipulação invalida todo o processo de posicionamento de código realizado pelo compilador. É importante lembrar que, como a técnica de desmontagem não é especulativa, alguns trechos de código podem não ter sido desmontados, e portanto não serão atualizados. Em alguns casos isto pode

[4] A noção de *macro blocos* é a generalização do conceito de blocos básicos aplicada aos próprios blocos básicos. Basicamente, um macro bloco compreende N blocos básicos da mesma forma que um bloco básico compreende n instruções.

[5] A *imagem executável* de um código executável é composta pelos segmentos que contêm instruções e são efetivamente carregados em memória para possibilitar sua execução.

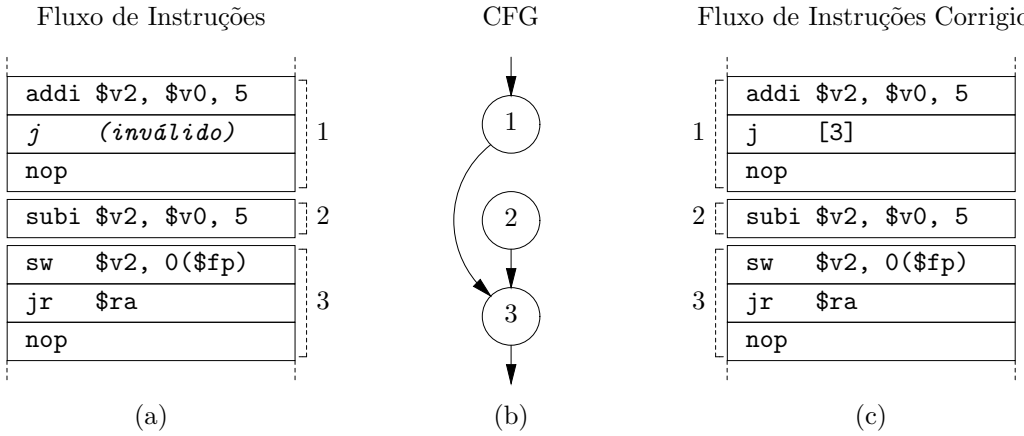

Figura 6.3: Visão geral do ajuste de endereços. Durante o estágio de consolidação, as referências das instruções de desvio são verificadas para evitar que o binário gerado contenha referências inválidas. O fluxo de instruções ilustrado em (a) contém uma destas referências inválidas no bloco básico 1. Com o auxílio do CFG em (b), o módulo de consolidação verifica o endereço virtual correto do bloco 3, ajustando a instrução do bloco 1 que o referencia, resultando no fluxo corrigido de instruções ilustrado em (c).

causar a geração de uma imagem executável inválida, sendo que a solução definitiva permanece como um trabalho futuro (Seção 8.1.3). Além disso, este módulo é responsável pela *segmentação* da imagem executável, isto é, dividir adequadamente, e se necessário, a imagem executável em múltiplos segmentos para que a imagem resultante possa ser corretamente carregada para a memória e então executada.

O processo de ajuste de endereços é ilustrado na Figura 6.3. Antes de iniciar o processo de segmentação, o módulo de consolidação analisa todos os blocos básicos, através do mapeamento virtual de memória, verificando se os destinos referenciados pelas instruções de desvio estão corretos. O fluxo de instruções ilustrado em (a) contém uma destas referências inválidas no bloco básico 1, na instrução de desvio incondicional `j`[6]. Com o auxílio do CFG, o módulo de consolidação determina o endereço virtual correto do destino (o bloco básico 3) e atualiza a instrução, resultando no fluxo de instruções corrigido ilustrado em (c). Vale lembrar que este processo é dependente de arquitetura, visto que é necessário inspecionar e manipular uma instrução de máquina, além de garantir a semântica do programa. No caso da arquitetura MIPS, o módulo deve ainda verificar se a instrução de desvio sendo manipulada será capaz de armazenar a distância do salto.

A Figura 6.4 ilustra a segmentação realizada durante o processo de consolidação. Tomando o mapeamento virtual de memória como ponto de início, em (a), o módulo de consolidação varre seqüencialmente de 0 até M, em busca de lacunas. Quando uma lacuna é encontrada, um novo segmento é criado para armazenar a região contígua encontrada, como por exemplo o segmento S_1 em (a) movido para uma posição adjacente ao segmento S_0, em (b). Este processo segue até que o módulo atinja o endereço virtual M, resultando em um agrupamento de n segmentos que compreendem o intervalo contíguo de 0 a m, sendo $m \leq M$. No contexto deste trabalho, cada um dos segmentos $S_0, S_1, \ldots, S_n$ será, efetivamente, um segmento no

[6]A instrução `j` é o mnemônico para *jump* (salto), e é usada para realizar um desvio incondicional dentro de segmentos de 256 MiB.

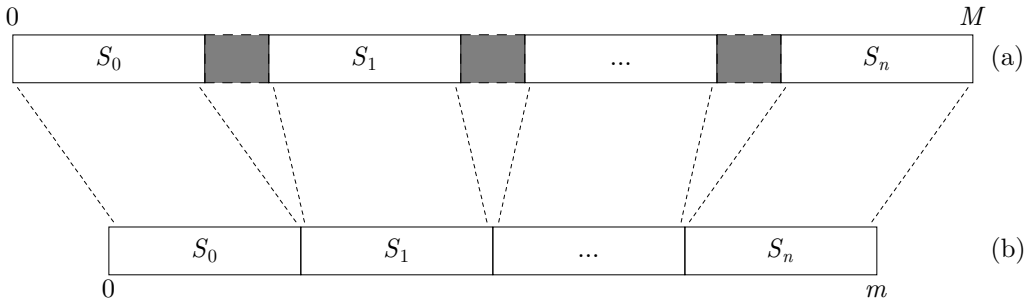

Figura 6.4: Visão geral do processo de segmentação. O módulo de consolidação identifica os segmentos $S_0, S_1, \ldots, S_n$ no mapeamento virtual de memória que compreende os endereços virtuais de 0 a M, em (a), removendo as lacunas que porventura existam (regiões escurecidas da Figura) entre eles, e os agrupa em uma seqüência de segmentos de tamanho m, em (b), sendo $m \leq M$.

arquivo ELF gerado.

A Figura 6.4 ilustra ainda como um arquivo ELF gerado é carregado para a memória: cada um dos segmentos consolidados em (b) é carregado considerando-se o posicionamento original, antes da consolidação, como ilustrado em (a). Pode-se também entender como 0 a M sendo o mapeamento em memória da imagem executável do código, enquanto que de 0 a m como sendo o mapeamento consolidado em arquivo.

6.3.2 Criação de Arquivo-objeto

O processo usual de geração de arquivos-objeto emprega um *ligador de código*[7] para combinar diversos arquivos-objeto gerados pelo compilador em um único, seguido por estágios de relocação de endereços e referências simbólicas [46]. Mesmo no caso de ferramentas que não tenham acesso ao código fonte ou demais informações do contexto de compilação, o processo é similar ao realizado pelos ligadores de código, visto que parte dos dados do arquivo original são combinados com as possíveis modificações realizadas.

Considerando a abordagem modular adotada, deve-se assumir o mínimo possível a respeito do formato do arquivo-objeto que será gerado. A ferramenta deve, portanto, passar todas as informações disponíveis para o módulo de criação, ficando a critério deste decidir o que será efetivamente usado. Dentre as informações disponíveis, pode-se citar o CFG, o mapeamento de memória virtual, os segmentos resultantes do estágio de consolidação, e os segmentos do arquivo de entrada que não foram modificados. A Seção 7.3.2 apresenta características específicas da criação de arquivos-objeto ELF, foco do trabalho proposto.

O resultado do módulo de criação é um novo arquivo-objeto no formato implementado pelo módulo, contendo tanto segmentos não-modificados do arquivo original quanto segmentos possivelmente modificados pelos módulos de manipulação.

[7]Em Inglês, *link-editor*.

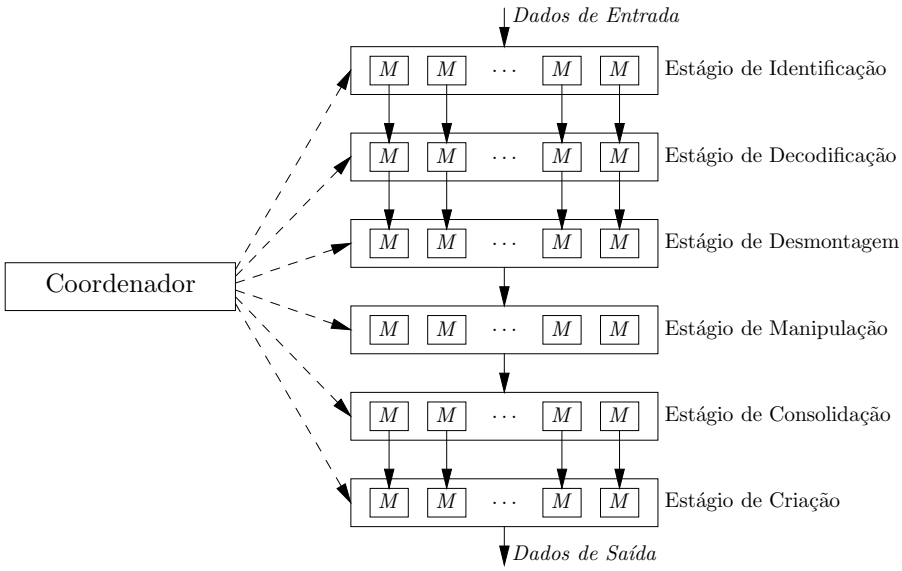

Figura 6.5: Visão geral da coordenação de estágios. O coordenador invoca seqüencialmente módulos de identificação conhecidos. Caso algum dos módulos de identificação tenha reconhecido o formato de dados, ele indica quais módulos de decodificação e desmontagem são apropriados, de forma que o coordenador os invoca diretamente. Após a desmontagem, o coordenador invoca cada um dos módulos de manipulação, na ordem especificada pelo usuário. Finalmente, o coordenador invoca o módulo de consolidação, que de forma semelhante ao módulo de identificação indica o módulo correto de criação, após realizar o ajuste de endereços e segmentação. O coordenador então invoca o módulo de criação, como último estágio, criando o arquivo-objeto final.

6.4 Coordenação de Estágios

O coordenador provê o funcionamento básico da infra-estrutura proposta na medida em que determina a ordem dos estágios, intermedia o fluxo de dados entre os estágios, gerencia as estruturas de dados (como por exemplo o CFG), entre outros. Conceitualmente, a coordenação de estágios não é realizada por um módulo como os apresentados nas Seções anteriores, mas pode ser melhor entendido como tal.

A Figura 6.5 ilustra o funcionamento do coordenador de estágios. O primeiro estágio é o de identificação dos dados de entrada, no topo da Figura. Nele, os módulos de identificação conhecidos são aplicados um após o outro até que um deles indique suportar o formato dos dados, encerrando a execução deste estágio. Na eventualidade de nenhum módulo reconhecer o formato de dados de entrada, a execução da ferramenta deve ser interrompida. Após a identificação, tem início o estágio de decodificação dos dados de entrada. Diferentemente do primeiro estágio, o coordenador apenas invoca o método correspondente ao estágio de decodificação, sendo que este foi definido pelo módulo de identificação. Esta abordagem possibilita a generalização do coordenador, tornando-o independente dos detalhes de formatos de dados de entrada, arquiteturas ou sistemas operacionais.

Analogamente ao estágio de decodificação, o coordenador invoca o módulo adequado para a desmontagem, também definido no primeiro estágio. Após a desmontagem, o coordenador invoca seqüencialmente cada um dos módulos de manipulação, que recebe uma referência ao CFG gerado no estágio de desmontagem. Ao final do

estágio de manipulação, o coordenador invoca o módulo de consolidação, responsável pelo ajuste de endereços e também pela segmentação da imagem executável. Além disso, o módulo de consolidação definirá, de forma análoga ao estágio de identificação, qual módulo de criação é apropriado para o formato dos dados de saída. Finalmente, coordenador invoca o módulo de criação no último estágio, encerrando o processo como um todo.

7 ESTUDO DE CASO

ReMIPS é uma ferramenta de código livre e aberta[1] criada com o objetivo de validar as abordagens e soluções propostas neste trabalho, e também identificar os pontos fracos das mesmas. Ela contempla todos os estágios descritos, e foi implementada na linguagem C. Tanto a implementação quanto os experimentos foram realizados sobre o sistema operacional FreeBSD [47], derivado do BSD[2]/UNIX, executando sobre a arquitetura Intel `80x86`.

Este Capítulo está organizado da seguinte forma. A Seção 7.1 detalha os módulos implementados que compõem o estágio de entrada de dados, especificamente os módulos de identificação, decodificação e desmontagem. O estágio de manipulação da ferramenta é composto por apenas um módulo, descrito na Seção 7.2. A Seção 7.3 detalha os módulos de consolidação e criação, que compõem o estágio de saída de dados. Os resultados obtidos são apresentados e detalhados na Seção 7.4, encerrando o Capítulo.

7.1 Estágio de Entrada de Dados

A arquitetura-alvo escolhida como estudo de caso para este trabalho é composta pelo formato de arquivos-objeto ELF e arquitetura de processadores MIPS. Especificamente, a arquitetura-alvo adotada é ELF32/MIPS, capaz apenas de lidar com arquivos-objeto ELF com capacidade de endereçamento de 32 bits. Durante a implementação da ferramenta, não foram encontrados casos onde uma maior distinção entre sistemas operacionais se fizesse necessária, visto que o formato ELF é adotado (ou pelo menos suportado) pela maior parte dos sistemas UNIX ou semelhantes ao UNIX disponíveis.

7.1.1 Módulo de Identificação

Os arquivos-objeto ELF são identificados pelo método de assinatura, como detalhado na Seção 6.1.1. A Figura 7.1 ilustra o número mágico adotado no formato ELF [34]. Como ilustrado na Figura, o número é composto pelos bytes `0x7F`, `0x45`, `0x4C`, e `0x46`, nesta ordem e estando o primeiro byte localizado exatamente no início do arquivo, juntos formando a palavra de 32 bits `0x7F454C46` em ordem de bytes *big-endian*. Os últimos três bytes representam os caracteres `E`, `L`, e `F`, respectivamente, no conjunto ASCII de caracteres.

[1] ReMIPS está licenciada sob a GNU GPL (*General Public License*), e será disponibilizada em `http://rnsanchez.wait4.org/remips/`.

[2] *Berkeley Software Distribution*

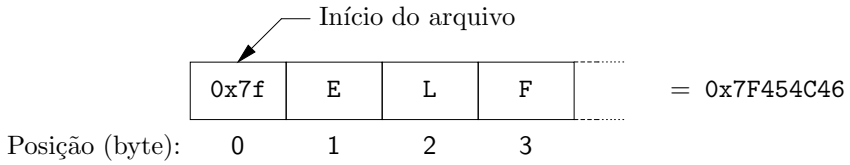

Figura 7.1: "Número mágico" adotado no formato ELF. A palavra de 32 bits 0x7F454C46 (em hexadecimal) é formado pelos 4 bytes 0x7F, 'E' (0x45), 'L' (0x4C), e 'F' (0x46), na ordem de bytes *big-endian*, e deve estar localizado exatamente no início do arquivo.

Além do número mágico, a especificação ELF prevê um mínimo de 12 bytes adicionais (totalizando 16 bytes) que contêm informações necessárias para a completa identificação do arquivo, como por exemplo ordem de bytes adotada, arquitetura de processador, capacidade de endereçamento, entre outras. Estes 16 bytes formam o bloco de identificação ELF (*ELF Identification*, em Inglês), constituindo o atributo `e_ident` do cabeçalho principal (detalhado na Seção 2.1.1).

A primeira etapa de identificação de um arquivo ELF compreende os seguintes passos, todos realizados sobre o bloco de identificação ELF (o atributo `e_ident` do cabeçalho principal):

1. Verificar a assinatura do arquivo ELF que deve, necessariamente, estar localizada exatamente no início do arquivo (byte 0 do arquivo).

2. Conferir a versão da especificação ELF que o arquivo atende, indicado no byte 6 do atributo `e_ident`. A especificação prevê apenas um valor simbólico válido para este atributo, `EV_CURRENT`.

3. Verificar se a classe do arquivo, indicada no byte 4, é válida e a ferramenta é capaz de manipulá-lo. A classe do arquivo determina sua capacidade de representação, podendo ser `ELFCLASS32` ou `ELFCLASS64`, denotando uma capacidade de endereçamento de 32 ou 64 bits, respectivamente. Com esta informação, a ferramenta pode determinar o tamanho esperado do cabeçalho principal para que possa carregá-lo adequadamente, em função da diferença de capacidade de cada classe.

4. Verificar se a ordem de bytes usada para representar os elementos específicos para o processador, indicada no byte 5, é válida e suportada por algum dos módulos de decodificação, podendo ser `ELFDATA2LSB` ou `ELFDATA2MSB`, significando ordem de bytes *little-endian* e *big-endian*, respectivamente. Este passo é fundamental, pois parte do próprio cabeçalho principal estará codificado de acordo com a ordem especificada.

A segunda e última etapa de identificação considera informações presentes nos demais atributos do cabeçalho principal, e compreende os seguintes passos:

1. Examinar o conteúdo do atributo `e_type`, que indica o tipo de arquivo dentre os previstos para o formato ELF. A ferramenta procura especificamente o valor simbólico `ET_EXEC`, empregado em programas executáveis comuns e que são o foco deste trabalho.

2. Verificar novamente a versão da especificação ELF que o arquivo atende, desta vez indicada no atributo `e_version`. A especificação só prevê um valor simbólico válido para este atributo, `EV_CURRENT`.

3. Verificar se a arquitetura do processador é suportada por algum dos módulos de desmontagem, especificada no atributo `e_machine`. A ferramenta implementada procura um dentre os valores simbólicos `EM_MIPS` e `EM_MIPS_RS4_BE`.

Se o módulo de identificação concluir todos os passos com sucesso, ele pode assumir que trata-se de um arquivo-objeto ELF válido. Apesar disto, os demais módulos verificam a consistência das informações utilizadas para evitar um processamento incorreto dos dados.

7.1.2 Módulo de Decodificação

Os módulos de decodificação são especialmente necessários quando deseja-se analisar um arquivo-objeto ELF com representação de bytes diferente da suportada pela arquitetura hospedeira, onde a ferramenta encontra-se em execução. Para isso, um módulo capaz de decodificar o conteúdo de arquivos ELF tanto na ordem de bytes *little-endian* quanto *big-endian* foi implementado.

Este módulo é capaz de decodificar tanto os cabeçalhos definidos no formato ELF, sendo que cada um possui uma estrutura própria, quanto blocos contíguos, como é o caso dos segmentos contendo instruções. No caso da arquitetura MIPS, o módulo de decodificação converte segmentos contíguos de acordo com o tamanho de palavra especificado no atributo `sh_addralign` ou `p_align` do respectivo cabeçalho de seção ou programa.

O funcionamento do módulo é bastante simples: uma vez identificado que a ordem de bytes empregada nos dados de entrada é diferente da suportada pela arquitetura hospedeira, o módulo realiza a conversão apropriada.

7.1.3 Módulo de Desmontagem

O módulo de desmontagem implementa a técnica recursiva estendida, detalhada no Capítulo 5, e é capaz de decodificar todas as instruções do conjunto de instruções MIPS32 [39; 42]. Com os segmentos que contêm instruções devidamente carregados no mapeamento virtual de memória, o módulo inicia a desmontagem no endereço virtual indicado no atributo `e_entry` do cabeçalho principal.

Para garantir a consistência de funcionamento da ferramenta, o módulo inspeciona as instruções em menor ou maior grau, dependendo de sua importância no processo de desmontagem. De uma forma geral, apenas as instruções de desvio são analisadas completamente, sendo que as demais são decodificadas apenas minimamente, isto é, o suficiente para determinar a instrução de forma não-ambígua, e também para identificar instruções desconhecidas ou inválidas. No caso da arquitetura MIPS, fabricantes podem definir instruções que não estejam presentes no conjunto de instruções MIPS32 ou MIPS64, conhecidas como *extensões de aplicação específica*[3], como é o caso do conjunto de instruções MIPS16e.

A arquitetura MIPS apresenta diferentes formatos de instruções de desvio, sendo que o destino pode ser expressado de forma relativa, alinhada, ou absoluta, enquanto que o desvio em si pode ser condicional ou incondicional. Quanto mais próximo do

[3] *Application-specific extensions*, em Inglês.

total de combinações possíveis de desvios o módulo de desmontagem for capaz de analisar, maior será a cobertura de desmontagem. O módulo implementado é capaz de analisar apenas os desvios cujos destinos podem ser computados estaticamente, conhecendo-se apenas, além da própria instrução, o seu endereço virtual.

As instruções de desvio relativo, como o nome sugere, são aquelas que trazem consigo uma distância a ser seguida caso o desvio seja tomado. No caso da arquitetura MIPS, esta distância é informada no campo `offset` da instrução (ilustrado na Figura 3.5 da Seção 3.4). Este campo de 16 bits (com sinal) representa a distância em *palavras*, devendo, portanto, ser multiplicado por 4 para gerar um valor de 18 bits em bytes. O resultado é então adicionado ao endereço da instrução seguinte ao *delay slot*, determinando o endereço destino efetivo. Por exemplo, tomando-se uma instrução de desvio relativo com `offset` O no endereço virtual M_i, seu destino é dado por $M_{i+2} + O \times 4$. A distância relativa que pode ser expressada em um desvio relativo é de, portanto, ±128 KiB.

Já as instruções de desvio alinhado expressam seu destino sob a forma de um deslocamento, também em palavras, dentro de um segmento de 256 MiB. O valor especificado no campo `address` da instrução é multiplicado por 4, gerando uma palavra de 28 bits, e então combinado com os 4 bits mais significativos do registrador especial `$pc` (*Program Counter*, contador de programa em Inglês) para formar o endereço destino, agora com 32 bits. Finalmente, os desvios absolutos são realizados tomando-se diretamente o valor contido em um dos registradores de propósito geral do processador como destino. Por causa desta característica, desvios deste tipo não podem ter seu destino computados facilmente de forma estática, como discutido na Seção 3.2, não sendo suportados pela ferramenta ReMIPS.

No caso específico da ferramenta implementada, tendo a arquitetura MIPS como estudo de caso, optou-se por popular o CFG diretamente com as instruções em seu formato codificado, visto que, no caso da arquitetura MIPS, lidar diretamente com as instruções codificadas é relativamente simples, além da maior eficiência de armazenamento em comparação com outras estruturas de dados [48]. Contudo, esta decisão restringe a aplicabilidade das técnicas de otimização, já que elas precisam atentar para detalhes de arquitetura.

7.2 Estágio de Manipulação

Para o estágio de manipulação de instruções, optou-se por implementar um módulo capaz de aplicar a técnica de otimização denominada *delayed branch optimization* [49; 50; 51]. Esta técnica é aplicável em arquiteturas que possuam o *delay slot*, como é o caso da arquitetura MIPS, e seu principal objetivo é mover instruções para que ocupem o *delay slot* caso ele não esteja sendo aproveitado.

Alguns compiladores (como o GCC), quando não instruídos a otimizarem o código a ser gerado, incluem instruções no *delay slot* que não realizam computação útil, reconhecidas pelo mnemônico `nop`[4]. A arquitetura MIPS não prevê uma instrução específica para este caso, existindo apenas a pseudo-instrução[5] `nop`. Esta é

[4] O mnemônico `nop` é a contração de *no-operation* (em Inglês), e instrui o processador a não realizar operação alguma nas arquiteturas que implementam tal instrução.

[5] Pseudo-instrução é aquela que não é prevista pelo conjunto de instruções da arquitetura, mas é oferecida pelo montador para maior conveniência do programador. No caso da arquitetura MIPS, `nop` e `move` são exemplos de pseudo-instruções.

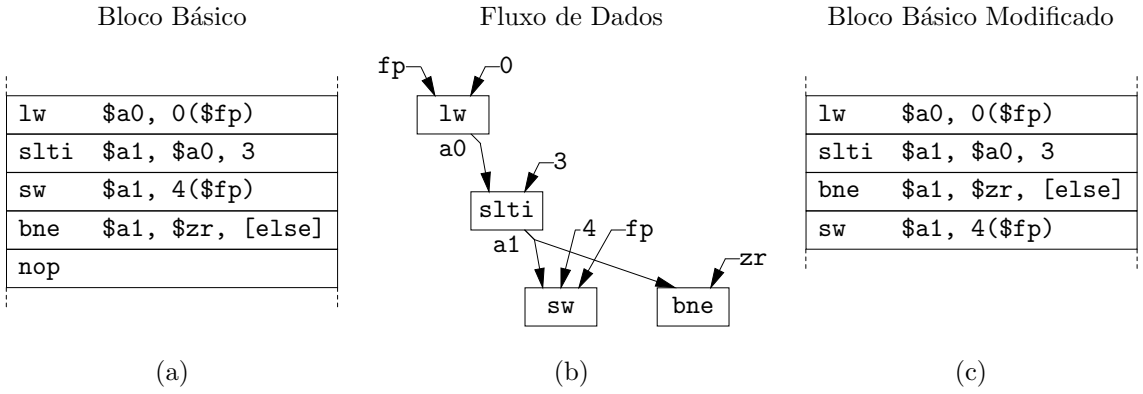

Figura 7.2: Técnica de otimização *delayed branch optimization*. Ao identificar que o bloco básico (a) apresenta um *delay slot* sub-utilizado, o módulo de otimização consulta o DFG (b), para determinar qual instrução de (a) pode ser movida a fim de ocupar o *delay slot*. Pelo DFG, o módulo verifica que a instrução `sw` pode ser movida, resultando no bloco básico modificado (c).

representada pela instrução `sll $zr, $zr, 0`[6], que instrui o processador a deslocar logicamente à esquerda (*shift logical left*, em Inglês) o registrador especial `$zr` em 0 bits, atribuindo o resultado novamente ao registrador `$zr`. Em alguns casos, o compilador pode optar por emitir outras instruções inócuas, que no caso da arquitetura MIPS podem ser identificadas como operações cujo registrador destino seja o registrador especial `$zr`.

O módulo analisa cada bloco básico separadamente, sendo que em um primeiro momento ele tenta identificar se (i) o bloco possui $n \geq 3$ instruções, e (ii) o bloco básico contém uma instrução de desvio. A importância da primeira restrição se deve ao fato de a técnica efetivamente remover uma instrução do bloco, e de que toda instrução de desvio na arquitetura MIPS é seguida pelo *delay slot*. No caso de um bloco com apenas duas instruções, o resultado seria um bloco inválido contendo apenas uma instrução—a de desvio. A segunda restrição se deve pois apenas blocos contendo instruções de desvio terão um *delay slot*. O fato de todas as instruções do conjunto de instruções MIPS possuírem o mesmo tamanho permite a este módulo de otimização identificar os blocos básicos contendo uma instrução de desvio em tempo constante ($O(1)$). Em um bloco básico B contendo n instruções, sendo $n \geq 3$, basta ao módulo analisar a instrução B_{n-1}. Neste caso, o módulo verifica ainda se a última instrução do bloco é inócua.

A Figura 7.2 ilustra o funcionamento desta técnica de otimização, implementada pelo módulo de manipulação. Em um primeiro momento, o módulo atinge o bloco básico (a) que contém as instruções ilustradas na Figura, e verifica se este bloco atende à primeira restrição, quanto à quantidade de instruções. Em seguida o módulo verifica se a instrução B_{n-1} é uma instrução de desvio, e se o *delay slot* possui uma instrução inócua, como ilustrado em (b). Como a instrução B_{n-1} é de fato uma instrução de desvio, o módulo analisa o DFG (b) para verificar qual instrução pode ser movida para o *delay slot*. O DFG (b) indica que o resultado da instrução

[6] A instrução `sll $zr, $zr, 0`, quando codificada, resulta na palavra de valor zero.

`slti`[7] (registrador `$a1`) é usado pelas instruções `sw`[8] e `bne`[9]. Neste ponto o módulo identifica que estas instruções não são interdependentes e, portanto, podem ter sua ordem alterada. Como o objetivo é mover uma computação útil para o *delay slot*, o módulo verifica que a instrução `sw` pode ser posicionada no *delay slot*, gerando o bloco básico modificado (c).

7.3 Estágio de Saída de Dados

Cada formato de arquivo-objeto apresenta peculiaridades em seu processo de geração, que devem ser abstraídas do restante da ferramenta pelos módulos de consolidação e criação. Com base nisto, é importante que este estágio tenha acesso à maior quantidade de informações possível, o que, no caso da ferramenta implementada, implica em oferecer todos os dados disponíveis, tanto os de entrada (como por exemplo o arquivo-objeto inicial) quanto os resultados parciais.

No caso específico da ferramenta implementada, o mesmo formato de arquivos-objeto é adotado para tanto entrada quanto saída de dados, sendo então possível aproveitar parte dos dados de entrada para os dados de saída.

7.3.1 Módulo de Consolidação

Antes que o módulo de criação possa efetivamente criar um novo arquivo-objeto, é necessário gerar uma imagem do mapeamento de memória, pelo processo de consolidação. Como detalhado na Seção 6.3.1, o módulo de consolidação deve realizar o ajuste de endereços e também a segmentação, para gerar a nova imagem executável.

Para o primeiro processo, o módulo implementado percorre seqüencialmente o mapeamento virtual de memória, em busca de blocos contendo instruções de desvio. Este processo requer, portanto, que o módulo leve em consideração os detalhes da arquitetura. Como mencionado na Seção 7.1.3, para cada bloco B contendo n instruções, o módulo pode verificar diretamente se a instrução B_{n-1} é uma instrução de desvio, e neste caso computar o endereço de destino correto, com a ajuda do CFG, e atualizar a instrução. Ainda, durante a correção da instrução o módulo pode identificar que a instrução de desvio não pode representar adequadamente o destino, em função da capacidade de representação de cada tipo de desvio. Por exemplo, no caso da arquitetura MIPS, para corrigir um desvio condicional relativo (adequado para saltos relativos na faixa de ±128 KiB) pode ser necessário incluir de instruções adicionais. Contudo, como conseqüência desta operação pode ser necessário computar o destino, e para isso será necessário utilizar um registrador, o que pode conflitar com a alocação de registradores já realizada, alterando a semântica do código. Embora convencionou-se que o registrador `$at`[10] seja reservado para casos como este [41], a ferramenta não pode assumir que esta prática tenha sido adotada. Sendo assim, caso a ferramenta identifique que uma instrução de desvio não possa ser adequadamente corrigida, a ferramenta não tem escolha senão abortar

[7] `slti` é o mnemônico para *set on less than immediate*, onde a condição é avaliada e o resultado (0 ou 1) atribuído ao registrador destino especificado na instrução.

[8] `sw` é o mnemônico para *store word*, onde o conteúdo do registrador indicado é armazenado na posição de memória indicada na instrução.

[9] `bne` é o mnemônico para *branch on not equal*, desviando a execução para a posição indicada na instrução caso a condição seja satisfeita

[10] *Assembler Temporary*, em Inglês.

a execução, alertando o usuário do ocorrido.

Para a segunda etapa, que consiste na geração da nova imagem executável, o módulo implementado realiza ainda um processo de segmentação, devido à característica dos arquivos-objeto ELF de organizarem seu conteúdo sob a forma de segmentos. Para tanto, o mapeamento virtual de memória é novamente percorrido seqüencialmente, de forma que os trechos contíguos são identificados, cada um gerando um segmento individual.

7.3.2 Módulo de Criação

O módulo de criação implementado gera um novo arquivo-objeto ELF a partir dos dados gerados pelo módulo de consolidação, composto basicamente por segmentos que representam a imagem executável do código. Além disso, o módulo de criação utiliza parte das informações contidas no arquivo ELF de entrada, como por exemplo os segmentos de dados.

Durante a criação do novo arquivo, o módulo de criação adota as seguintes considerações.

Cabeçalho principal. A quantidade de cabeçalhos de programa (`e_phnum`) e posição (`e_phoff`), bem como a quantidade (`e_shnum`) e posição (`e_shoff`) dos cabeçalhos de seção deve refletir a quantidade de segmentos do novo arquivo. Se necessário, a ferramenta também atualiza o ponto de entrada do programa (`e_entry`), caso algum módulo de manipulação tenha alterado o bloco básico raiz a ponto de ser necessária a atualização do ponto de entrada.

Cabeçalhos de programa. Para que o novo arquivo ELF possa ser executado, a ferramenta precisa gerar novos cabeçalhos de programa condizentes com a nova disposição dos segmentos executáveis. Podem ser necessários múltiplos cabeçalhos de programa, devido a possíveis lacunas no endereçamento virtual, herdadas do arquivo original ou conseqüência das modificações realizadas. Os cabeçalhos que referenciam elementos externos, como por exemplo bibliotecas dinâmicas, são tratados de forma conservadora, sendo preservados no novo arquivo. A remoção de referências externas ou inclusão de novas requer uma análise mais completa dos efeitos colaterais por parte da ferramenta, especialmente no caso da remoção.

Cabeçalhos de seção. Analogamente aos cabeçalhos de programa, novos cabeçalhos de seção são gerados para as seções que porventura tenham sido modificadas. Seções que não contêm instruções executáveis são preservadas no novo arquivo, tendo apenas sua posição em arquivo (`sh_offset`) atualizada.

Segmentos de dados. Segmentos que não tenham sido modificados são preservados no novo arquivo, assim como seus respectivos cabeçalhos de seção. Dependendo das modificações realizadas, alguns segmentos podem ser consolidados em um único, ou ainda divididos em múltiplos segmentos, fazendo com que o arquivo ELF resultante tenha uma quantidade diferente de segmentos em comparação ao arquivo original.

Ordem de bytes. De acordo com a arquitetura-alvo, a ordem de bytes pode ser preservada, em comparação ao arquivo original, ou adaptada, caso a ordem da arquitetura-alvo seja diferente da adotada no arquivo original. Neste caso,

além das regiões modificadas, a ferramenta deve adaptar a ordem de bytes dos demais componentes do arquivo, ao invés de simplesmente copiar a partir do arquivo de entrada.

7.4 Resultados

A ferramenta implementada foi submetida a uma avaliação experimental, tomando como entrada diferentes programas, cada um modelando um dos seguintes cenários:

A. Implementação do algoritmo de ordenação *bubble-sort* sobre um vetor com quantidade arbitrária de elementos.

B. Construção do tipo `if...then...else`, com uma expressão simples como condição.

C. Implementação de um algoritmo que calcula a soma de verificação (*checksum*, em Inglês) para cabeçalhos de pacotes IP[11].

D. Um programa que não realiza nenhuma operação.

E. Construção do tipo `switch` (*expr*) com uma quantidade arbitrária de alternativas.

Para cada um dos cenários descritos, foram gerados diferentes programas com o compilador GCC[12] [52], sendo um para cada conjunto pré-definido de otimizações oferecido pelo compilador. Especificamente, o GCC oferece quatro níveis de otimização (0–3, sendo 0 nenhuma otimização, e 3 todas as otimizações possíveis) e um conjunto adicional (`s`) que inclui apenas técnicas de otimização que reduzam, ou pelo menos não aumentem, o tamanho do código gerado. Considerando que o módulo de manipulação implementa a técnica de aproveitamento dos *delay slots*, o compilador foi instruído[13] a nunca aplicar esta técnica, pois como ela faz parte de todos os conjuntos de otimização (exceto do nível 0), o código gerado não apresentaria possibilidades de otimização.

Os cenários são propositalmente reduzidos pois cada caso deve ser verificado manualmente em sua totalidade. Isto é, os binários que serão usados como entrada de dados são inspecionados, assim como os binários gerados pela ferramenta. Os resultados parciais de cada estágio são conferidos com os dados gerados, bem como o CFG e DFG gerados. Ainda, as oportunidades de otimização devem ser identificadas, e então verificar quais foram corretamente aproveitadas e quais foram corretamente ignoradas, já que nem todos os casos podem ser aproveitados sem uma maior reestruturação do código.

Os resultados obtidos são apresentados na Tabela 7.1. A coluna *Cenário* indica a qual dos cinco cenários distintos pertencem os resultados apresentados. A coluna *Otimização* indica o nível de otimização empregado, sendo que 0 para nenhuma otimização, 3 para todas as otimizações possíveis, e `s` para o conjunto voltado à

[11] *Internet Protocol* em Inglês, um protocolo do nível de rede da pilha TCP/IP.

[12] GCC versão 3.4.4 com arquitetura-alvo ELF32/UNIX/MIPS.

[13] No caso do GCC, a diretiva `-fno-delay-slots` o instrui a não aplicar a técnica *delayed branch optimization*.

Tabela 7.1: Resultados da aplicação da técnica de otimização. O módulo de manipulação, que implementa a técnica *delayed branch optimization*. Para cada cenário, foram gerados binários executáveis contemplando cada um dos níveis de otimização pré-definidos pelo compilador, além de instruí-lo a não aplicar a técnica equivalente à implementada pelo módulo. A Tabela apresenta a quantidade de casos efetivamente aproveitados, em comparação com o total de casos para uma determinada quantidade de instruções.

Cenário	Otimização	Instruções	Casos	Aproveitados
A	0	59	6	3
A	1	28	5	1
A	2	30	5	5
A	3	30	5	5
A	`s`	30	5	4
B	0	20	3	2
B	1	5	1	1
B	2	5	1	1
B	3	5	1	1
B	`s`	5	6	1
C	0	75	8	5
C	1	42	8	3
C	2	41	8	6
C	3	54	10	6
C	`s`	41	8	6
D	0	9	1	1
D	1	3	1	1
D	2	3	1	1
D	3	3	1	1
D	`s`	3	1	1
E	0	36	6	4
E	1	21	5	4
E	2	23	5	4
E	3	23	5	4
E	`s`	21	5	4

geração de código compacto. O total de instruções existentes nos programas gerados para cada combinação de cenário e conjunto de otimizações é apresentado na coluna *Instruções*. A quantidade de *delay slots* existentes em cada caso é apresentada na coluna *Casos*, sendo que deste total apenas a quantidade apresentada na coluna *Aproveitadas* foram realmente ser aproveitadas, em função das restrições indicadas DFG.

Como pode-se verificar pela Tabela, nem todas as oportunidades puderam ser efetivamente aproveitadas. A inspeção manual revelou que alguns casos foram prejudicados devido a uma alocação de registradores muito pobre, por parte do compilador, em especial nos casos sem otimização (nível 0). Nestas condições, um módulo de realocação de registradores mostraria-se efetivo.

8 CONCLUSÃO E TRABALHOS FUTUROS

As áreas relacionadas à re-engenharia de código vêm ganhando crescente atenção, tanto da comunidade científica quanto da indústria. Diversas técnicas e soluções são propostas visando um melhor aproveitamento dos recursos disponíveis, sejam eles armazenamento, energia, ou tempo, sendo que resultados significativos têm sido obtidos em áreas que até atualmente tinham sido pouco exploradas [53].

A principal contribuição deste trabalho é uma infra-estrutura modular e extensível que permite estudar os efeitos da aplicação de técnicas de re-engenharia de código, atuando diretamente sobre código executável. A abordagem é orientada a estágios, compostos por módulos que realizam atividades bem definidas, aumentando o espectro de aplicação da infra-estrutura proposta.

ReMIPS é uma ferramenta que implementa a infra-estrutura proposta, estando inicialmente focada na manipulação de binários gerados para a arquitetura-alvo ELF/MIPS. Ela ainda implementa um módulo de otimização como forma de validação da solução proposta. Os resultados obtidos encorajam a realização de trabalhos adicionais que complementem ou estendam as soluções propostas.

8.1 Trabalhos Futuros

Durante a implementação da ferramenta ReMIPS, diversas possibilidades de trabalhos futuros foram identificadas. As Seções a seguir detalham as vertentes consideradas mais relevantes.

8.1.1 Linguagem de Otimização

Algumas abordagens visam facilitar o estudo e desenvolvimento de técnicas de otimização [19; 20; 28; 29]. Contudo, estas abordagens não amenizam suficientemente a complexidade de implementação das técnicas, sendo que grande parte do problema deve ser tratado pelo desenvolvedor. Ainda, uma quantidade muito expressiva de técnicas de otimização pode ser encontrada na bibliografia especializada, ainda que nem todas sejam efetivamente implementadas em compiladores como o GCC. Parte deste distanciamento entre o estado-da-arte e o que efetivamente é disponibilizado pelas ferramentas se deve não apenas à complexidade de implementação das técnicas em si, mas também ao grande acréscimo de esforço necessário para validar e analisar o impacto da técnica adicionada com relação às demais, já implementadas.

Com base nisto, um mecanismo que permita ao desenvolvedor ou pesquisador descrever uma técnica de otimização com maior nível de abstração, similar ao apre-

sentado em [28], será de grande valia. Especificamente, uma linguagem com alto nível de abstração e manutenibilidade, que permita a descrição de técnicas de otimização, e então alimente uma ferramenta capaz de converter estas descrições em implementação. Isto poderá alavancar estudos bastante efetivos das interações entre as técnicas [2; 3], aprimoramento das técnicas existentes, e maior rapidez no desenvolvimento de novas técnicas, por permitir que um maior número de pesquisadores se envolvam com técnicas de otimização, servindo de base para uma grande quantidade de trabalhos adicionais.

8.1.2 Representação Intermediária

Um ponto bastante pertinente, em especial no contexto de conversão entre conjuntos de instruções (tradução binária), é a forma de representação intermediária para as instruções. Uma possibilidade é a utilização de linguagens do tipo RTL[1], onde uma operação é subdividida em diversas operações menores e simples. Porém, devido à tendência de operações relativamente simples na linguagem de origem (linguagem de montagem, por exemplo) gerarem muitas operações na linguagem RTL, problemas que em princípio não apresentam uma complexidade muito significativa podem se tornar intratáveis quando convertidos para uma representação em RTL.

Trabalhos como [25; 43; 48; 54] apresentam outras possibilidades para a representação de instruções de uma forma mais genérica.

8.1.3 Técnicas de Desmontagem

Ferramentas como ReMIPS, que adotam uma abordagem estática, podem apresentar resultados pouco satisfatórios em função de características do código sendo analisado. Durante a implementação da ferramenta, diversos binários MIPS não puderam ser plenamente analisados em função de características freqüentemente encontradas em código gerado atualmente, mas que por causa disso só podem ser adequadamente analisados com o emprego de técnicas como a análise simbólica, discutida na Seção 3.2. A inclusão de execução simbólica, por exemplo, pode permitir a análise de código auto-modificável e também polimórfico (código capaz de gerar uma representação equivalente, mas distinta, de si próprio), por exemplo.

Analogamente, a incorporação de técnicas especulativas de desmontagem podem ampliar o espectro de aplicação da ferramenta para binários que, por qualquer motivo, não apresentam uma estrutura adequada, ou passaram por um estágio de manipulação para dificultar sua análise [8], ou ainda um binário danificado ou gerado incorretamente.

8.1.4 Tradução Binária

O tópico de tradução binária tem recebido muita atenção da comunidade científica [16; 17; 23; 24; 25; 27], e até mesmo da indústria [26].

Com os módulos apropriados, a ferramenta ReMIPS pode realizar a tradução binária, com complexidade limitada pela capacidade dos módulos em si, tanto entre arquiteturas diferentes, como por exemplo da arquitetura Intel `80x86` para a arquitetura MIPS e vice-versa, quanto para variações da própria arquitetura, como por exemplo entre os processadores MIPS64, MIPS32, e MIPS16e da arquitetura MIPS.

[1] *Register Transfer Language*

8.1.5 Variações do Formato ELF

O formato ELF é empregado em diferentes tipos de binários contendo código executável, como por exemplo bibliotecas compartilhadas. Suportar tantas variações de um formato projetado para ser extremamente flexível contitui, por si só, um grande desafio.

Em especial, a grande quantidade de informações simbólicas que podem estar presentes em um arquivo ELF pode dificultar a interoperabilidade da ferramenta caso haja, por exemplo, uma tradução entre diferentes formatos de arquivo-objeto, somado à complexidade de preservar a consistência de uma variedade muito grande de informações.

REFERÊNCIAS

[1] BACKUS, J. The history of FORTRAN I, II, and III. *ACM SIGPLAN Notices*, ACM Press, New York, NY, EUA, v. 13, n. 8, p. 165–180, 1978. ISSN 0362-1340. Disponível em: <http://portal.acm.org/citation.cfm?id=808380>. Acesso em: 9 de novembro de 2006.

[2] SAAVEDRA, R. H.; SMITH, A. J. Performance characterization of optimizing compilers. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Piscataway, NJ, EUA, v. 21, n. 7, p. 615–628, 1995. ISSN 0098-5589. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=392982>. Acesso em: 31 de agosto de 2006.

[3] PAN, Z.; EIGENMANN, R. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*. IEEE Computer Society, 2006. p. 322–333. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1611551>. Acesso em: 31 de agosto de 2006.

[4] GCC, the GNU Compiler Collection. Disponível em: <http://gcc.gnu.org/>. Acesso em: 30 de agosto de 2006.

[5] DEBRAY, S. K. et al. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM Press, New York, NY, EUA, v. 22, n. 2, p. 378–415, 2000. ISSN 0164-0925. Disponível em: <http://portal.acm.org/citation.cfm?id=349233>. Acesso em: 30 de agosto de 2006.

[6] ZARETSKY, D. et al. Generation of control and data flow graphs from scheduled and pipelined assembly code. In: *Languages and Compilers for Parallel Computing*. Springer, 2006. (Lecture Notes in Computer Science, v. 4339), p. 76–90. ISBN 978-3-540-69329-1. Disponível em: <http://www.springerlink.com/content/r067666677huhp55/>.

[7] ELLISTON, B. *Studying Optimisation Sequences in the GNU Compiler Collection*. Canberra, Austrália: Australian Defence Force Academy, jun. 2005. 95 p. Disponível em: <https://www.air.net.au/~bje/elliston05.pdf>. Acesso em: 8 de maio de 2007.

[8] LINN, C.; DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. In: *CCS '03: Proceedings of the 10th ACM conference*

on Computer and communications security. New York, NY, EUA: ACM Press, 2003. p. 290–299. ISBN 1-58113-738-9. Disponível em: <http://portal.acm.org/citation.cfm?id=948149>. Acesso em: 14 de maio de 2007.

[9] CIFUENTES, C.; FRABOULET, A. Intraprocedural static slicing of binary executables. In: *Proceedings of the International Conference on Software Maintenance (ICSM 1997)*. IEEE Computer Society, 1997. p. 188–195. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=624245>. Acesso em: 11 de setembro de 2006.

[10] SCHWARZ, B.; DEBRAY, S.; ANDREWS, G. Disassembly of executable code revisited. In: *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE Computer Society, 2002. p. 45–54. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1173063>. Acesso em: 30 de agosto de 2006.

[11] KISS Ákos et al. Interprocedural static slicing of binary executables. In: *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*. IEEE Computer Society, 2003. p. 118–127. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1238038>. Acesso em: 31 de agosto de 2006.

[12] STITT, G.; VAHID, F. New decompilation techniques for binary-level coprocessor generation. In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2005)*. IEEE Computer Society, 2005. p. 547–554. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1560127>. Acesso em: 31 de agosto de 2006.

[13] KRUEGEL, C. et al. Static disassembly of obfuscated binaries. In: *Proceedings of the 13th USENIX Security Symposium*. USENIX Association, 2004. p. 255–270. Disponível em: <http://www.usenix.org/events/sec04/tech/kruegel.html>. Acesso em: 15 de outubro de 2006.

[14] KRUEGEL, C.; ROBERTSON, W.; VIGNA, G. Detecting kernel-level rootkits through binary analysis. In: *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*. Washington, DC, USA: IEEE Computer Society, 2004. p. 91–100. ISBN 0-7695-2252-1. Disponível em: <http://www.acsac.org/2004/papers/99.pdf>. Acesso em: 15 de outubro de 2006.

[15] KRUEGEL, C. et al. Automating mimicry attacks using static binary analysis. In: *Proceedings of the 14th USENIX Security Symposium*. Berkeley, CA, EUA: USENIX Association, 2005. p. 161–176. Disponível em: <http://www.usenix.org/events/sec05/tech/kruegel.html>. Acesso em: 15 de outubro de 2006.

[16] BRUENING, D.; GARNETT, T.; AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO 2003)*. [s.n.], 2003. p. 265–275. Disponível em: <http://dx.doi.org/10.1109/JPROC.2004.840302>. Acesso em: 1 de agosto de 2006.

[17] DUESTERWALD, E. Design and engineering of a dynamic binary optimizer. *Proceedings of the IEEE*, IEEE Computer Society, v. 93, p. 436–448, 2005. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1386661>. Acesso em: 20 de novembro de 2006.

[18] NANDA, S. et al. BIRD: Binary interpretation using runtime disassembly. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*. IEEE Computer Society, 2006. p. 359–370. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1611554>. Acesso em: 17 de junho de 2006.

[19] BENITEZ, M. E.; DAVIDSON, J. W. The advantages of machine-dependent global optimization. In: *Proceedings of the Conference on Programming Languages and System Architectures*. [s.n.], 1994. p. 105–124. Disponível em: <http://citeseer.ist.psu.edu/benitez94advantages.html>. Acesso em: 30 de agosto de 2006.

[20] BENITEZ, M. E.; DAVIDSON, J. *Target-specific Global Code Improvement: Principles and Applications*. Charlottesville, VA, EUA, 1994. Disponível em: <http://citeseer.ist.psu.edu/benitez94targetspecific.html>. Acesso em: 30 de agosto de 2006.

[21] LO, R. et al. Improving resource utilization of the MIPS r8000 via post-scheduling global instruction distribution. In: *MICRO 27: Proceedings of the 27th Annual International Symposium on Microarchitecture*. New York, NY, EUA: ACM Press, 1994. p. 148–152. ISBN 0-89791-707-3. Disponível em: <http://portal.acm.org/citation.cfm?id=192724.192745>. Acesso em: 30 de agosto de 2006.

[22] ROHOU, E. et al. *SALTO: System for Assembly-Language Transformation and Optimization*. Rennes, França, jun. 1996. Disponível em: <http://citeseer.ifi.unizh.ch/rohou96salto.html>. Acesso em: 8 de setembro de 2006.

[23] CIFUENTES, C.; MALHOTRA, V. Binary translation: Static, dynamic, retargetable? In: *Proceedings of the International Conference on Software Maintenance (ICSM 1996)*. IEEE Computer Society, 1996. p. 340–349. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=565037>. Acesso em: 30 de agosto de 2006.

[24] UNG, D.; CIFUENTES, C. Machine-adaptable dynamic binary translation. In: *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*. New York, NY, EUA: ACM Press, 2000. p. 41–51. Disponível em: <http://portal.acm.org/citation.cfm?id=351397.351414>. Acesso em: 31 de agosto de 2006.

[25] CIFUENTES, C. et al. *Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework*. Santa Clara, CA, EUA, jan. 2002. Disponível em: <http://research.sun.com/techrep/2002/abstract-105.html>. Acesso em: 22 de novembro de 2006.

[26] DEHNERT, J. C. et al. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In: *Proceedings of the International Symposium on Code Generation and Optimization*. Piscataway, NJ, EUA: IEEE Computer Society, 2003. p. 15–24. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1191529>. Acesso em: 17 de junho de 2006.

[27] UNG, D.; CIFUENTES, C. Dynamic re-engineering of binary code with run-time feedbacks. In: *Proceedings of the 7th Working Conference on Reverse Engineering*. Piscataway, NJ, EUA: IEEE Computer Society, 2000. p. 2–10. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=891447>. Acesso em: 30 de agosto de 2006.

[28] TJIANG, S. W. K.; HENNESSY, J. L. Sharlit–a tool for building optimizers. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*. New York, NY, EUA: ACM Press, 1992. p. 82–93. ISBN 0-89791-475-9. Disponível em: <http://doi.acm.org/10.1145/143095.143120>.

[29] WILSON, R. P. et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, ACM Press, New York, NY, EUA, v. 29, n. 12, dez. 1994. ISSN 0362-1340. Disponível em: <http://citeseer.ifi.unizh.ch/wilson94suif.html>. Acesso em: 30 de agosto de 2006.

[30] SRIVASTAVA, A.; WALL, D. W. *A Practical System for Intermodule Code Optimization at Link-Time*. Palo Alto, CA, EUA, dez. 1992. Disponível em: <http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-92-6.pdf>. Acesso em: 10 de agosto de 2006.

[31] LARUS, J. R.; BALL, T. Rewriting executable files to measure program behavior. *Software–Practice & Experience*, John Wiley & Sons, Inc., New York, NY, USA, v. 24, n. 2, p. 197–218, 1994. ISSN 0038-0644. Disponível em: <http://portal.acm.org/citation.cfm?id=181180.182418>. Acesso em: 14 de maio de 2007.

[32] HO, W. W.; CHANG, W.-C.; LEUNG, L. H. Optimizing the performance of dynamically-linked programs. In: *USENIX Winter*. Berkeley, CA, EUA: USENIX Association, 1995. p. 225–233. Disponível em: <http://www.usenix.org/publications//library/proceedings/neworl/ho.html>. Acesso em: 30 de agosto de 2006.

[33] LEVINE, J. *Linkers and Loaders*. 1. ed. Burlington, MA, EUA: Morgan Kaufmann, 1999. 256 p. (The Morgan Kaufmann Series in Software Engineering and Programming). ISBN 1-55860-496-0.

[34] TIS Committee. *Executable and Linking Format (ELF) Specification Version 1.2*. [S.l.], maio 1995. Disponível em: <http://x86.ddj.com/ftp/manuals/tools/elf.pdf>. Acesso em: 23 de novembro de 2006.

[35] COOPER, C.; MOORE, C. *HP-UX 11i Internals*. 1. ed. Upper Saddle River, NJ, EUA: Prentice-Hall, 2004. 432 p. ISBN 0-13-032861-8.

[36] Sun Microsystems Inc. *Linker and Libraries Guide*. Santa Clara, CA, EUA, nov. 2006. 364 p. (Solaris 10 Software Developer Collection). Disponível em: <`http://docs.sun.com/app/docs/doc/817-1984`>. Acesso em: 10 de maio de 2007.

[37] MCKUSICK, M. K.; NEVILLE-NEIL, G. V. *The Design and Implementation of the FreeBSD Operating System*. 1. ed. Boston, MA, EUA: Addison-Wesley, 2004. 720 p. ISBN 0-201-70245-2.

[38] NEVELN, B. *Linux Assembly Language Programming*. 1. ed. Upper Saddle River, NJ, EUA: Prentice-Hall, 2000. 272 p. ISBN 0-13-087940-1.

[39] MIPS Technologies. *Introduction to the MIPS32 Architecture*. 2.50. ed. Mountain View, CA, EUA, jul. 2005. 95 p. (MIPS32 Architecture For Programmers, v. 1). Disponível em: <`http://www.mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/doclibrary`>. Acesso em: 6 de agosto de 2006.

[40] CIFUENTES, C.; EMMERIK, M. V. Recovery of jump table case statements from binary code. In: *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*. Washington, DC, EUA: IEEE Computer Society, 1999. p. 192–199. ISBN 0-7695-0179-6.

[41] PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design*: The hardware/software interface. 3. ed. Burlington, MA, EUA: Morgan Kaufmann, 2004. 656 p. ISBN 1-55860-604-1.

[42] MIPS Technologies. *The MIPS32 Instruction Set*. 2.50. ed. Mountain View, CA, EUA, jul. 2005. 337 p. (MIPS32 Architecture For Programmers, v. 2). Disponível em: <`http://www.mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/doclibrary`>. Acesso em: 6 de agosto de 2006.

[43] CIFUENTES, C.; GOUGH, K. J. Decompilation of binary programs. *Software–Practice and Experience*, John Wiley & Sons, Inc., New York, NY, EUA, v. 25, p. 811–829, 1995. Disponível em: <`http://portal.acm.org/citation.cfm?id=213593.213604`>. Acesso em: 11 de setembro de 2006.

[44] MIPS Technologies. *Introduction to the MIPS64 Architecture*. 2.50. ed. Mountain View, CA, EUA, jul. 2005. 97 p. (MIPS64 Architecture For Programmers, v. 1). Disponível em: <`http://www.mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/doclibrary`>. Acesso em: 28 de maio de 2007.

[45] The IEEE and The Open Group. *`file`*. IEEE Std 1003.1, 2004 Edition. Disponível em: <`http://www.opengroup.org/onlinepubs/009695399/utilities/file.html`>. Acesso em: 7 de maio de 2007.

[46] GNU Linker `ld`. Disponível em: <`http://sourceware.org/binutils/docs-2.17/ld/`>. Acesso em: 8 de março de 2007.

[47] THE FreeBSD Project. Disponível em: <`http://www.freebsd.org/`>. Acesso em: 10 de junho de 2007.

[48] KOSCHKE, R.; GIRARD, J.-F.; WüRTHNER, M. An intermediate representation for reverse engineering analyses. In: *WCRE '98: Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*. Los Alamitos, CA, EUA: IEEE Computer Society, 1998. p. 241–250. ISBN 0-8186-8967-6. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/WCRE.1998.723194>. Acesso em: 30 de agosto de 2006.

[49] GROSS, T. R.; HENNESSY, J. L. Optimizing delayed branches. In: *MICRO 15: Proceedings of the 15th annual workshop on Microprogramming*. Piscataway, NJ, EUA: IEEE Computer Society, 1982. p. 114–120. Disponível em: <http://portal.acm.org/citation.cfm?id=800941>. Acesso em: 30 de agosto de 2006.

[50] MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. San Diego, CA, EUA: Academic Press, 1997. 532–547 p. ISBN 1-55860-320-4.

[51] SCOTT, M. L. *Programming Language Pragmatics*. San Diego, CA, EUA: Academic Press, 2000. 227–241 p. ISBN 1-55860-442-1.

[52] MIPS SDE Lite. Disponível em: <http://www.mips.com/products/softwaretools/software_tools/MIPS_SDE_Lite.php>. Acesso em: 28 de setembro de 2006.

[53] ARNOLD, M. et al. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, IEEE Computer Society, v. 93, p. 449–466, 2005. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1386662>. Acesso em: 16 de novembro de 2006.

[54] CIFUENTES, C. Structuring decompiled graphs. In: *CC '96: Proceedings of the 6th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 1996. p. 91–105. ISBN 3-540-61053-7. Disponível em: <http://portal.acm.org/citation.cfm?id=727463>. Acesso em: 1 de setembro de 2006.