Disassembly - Uma Fase Essencial para a Tradução e Interpretação Confiável de Binários*

Fabiano Ramos
ramosf@cos.ufrj.br

Edil S. T. Fernandes
edil@cos.ufrj.br

Relatório Técnico (Versão Preliminar)
Programa de Engenharia de Sistemas e Computação
COPPE - UFRJ
Cx. Postal 68.511
21945-970 - Rio de Janeiro - RJ, Brasil

Resumo

Este trabalho descreve o desenvolvimento da etapa *Disassembly* - uma importante componente da Tradução e Interpretação Binária. *Disassembly* é uma técnica de "engenharia reversa" que recupera a representação simbólica das instruções de um arquivo binário, no nosso caso, de programas em linguagem da máquina x86 controlada pelo Sistema Operacional Linux.

Duas técnicas estáticas de detecção e validação do maior número possível de instruções foram usadas inicialmente. Em seguida, empregamos uma técnica dinâmica, baseada em Interpretação Binária, identificamos os endereços alvejados por desvios indiretos (justamente um dos grandes desafios da recuperação das instruções de um programa binário), permitindo assim que as fronteiras de novos blocos básicos sejam delimitadas.

Além de especificar os algoritmos estáticos e dinâmico, o relatório exibe importantes características arquiteturais dos programas não numéricos do conjunto SPEC2000, indica os maiores obstáculos da Tradução e Interpretação Binária e lista promissores tópicos de pesquisa relacionados com nosso trabalho.

1 Introdução

Tradução Binária é o processo que converte o código binário de uma arquitetura para outra, visando possivelmente uma posterior Interpretação. Este processo pode ser estático ou dinâmico. No primeiro caso, o arquivo executável é convertido antes de ser interpretado na arquitetura hospedeira. Na tradução dinâmica, as instruções são convertidas à medida que são alcançadas pelo fluxo de controle do programa. Em ambos os casos, algumas características tornam esta tarefa bastante árdua, como por exemplo se o código for auto-modificável (*self-modifying code*).

Esta técnica tem sido amplamente utilizada no projeto de novos processadores, permitindo assim que o código binário existente seja reconhecido e obedecido na nova hospedeira. Por exempo, o processador Crusoe da Transmeta realiza a tradução binária do código x86 para as instruções muito longas de sua arquitetura VLIW, via *software*. Outra razão é a necessidade de se alterar a arquitetura interna de um processador sem perder a compatibilidade binária com os programas compilados anteriormente, como tem ocorrido com a família de processadores Pentium da Intel e com os recentes processadores da AMD. Nestes processadores, as instruções CISC x86 são traduzidas dinamicamente para as primitivas mais elementares de um *core* RISC. Neste caso, a tradução é levada a cabo por

*Pesquisa financiada pelo CNPq - Projeto BinTran 55.2091/02-2

decodificadores existentes no *hardware* subjacente destes processadores.

A obtenção da representação simbólica das instruções no código binário é então o primeiro passo para a Tradução e Interpretação de binários.

Disassembly é o processo de recuperação do correspondente programa em linguagem simbólica a partir do código binário. Em outras palavras, *Disassembly* é a técnica de "engenharia reversa" do que foi realizado pelo *Assembler*. Além de ser essencial para a Tradução e Interpretação Binária ela também é usada para a Otimização de código Executável.

Em máquinas do tipo CISC (*Complex Instruction Set Computers*), o *disassembly* de um programa é uma tarefa não trivial devido: (i) uma instrução pode iniciar e terminar em qualquer *byte* do código, sem levar em consideração se o endereço do *byte* coincide com o início de uma palavra; (ii) não é raro encontrarmos dados (*jump tables*, *bytes* de alinhamento, etc.) em áreas tipicamente reservadas para instruções; (iii) o tamanho das instruções não é fixo. Por exemplo, na arquitetura Intel x86 nem sempre podemos afirmar se um *byte* do executável faz parte de uma instrução ou de um dado.

Estes problemas aparecem com maior intensidade nas funções das bibliotecas, pois muitas vezes elas incluem trechos codificados diretamente em linguagem de máquina e são bastante otimizadas. Um *disassembler* deve ser capaz de lidar também com estas funções, já que os binários que serão traduzidos, interpretados ou otimizados normalmente são compilados estaticamente (i.e., as funções da biblioteca são introduzidas no executável).

Trabalhos recentes mostram que nem sempre pode-se recuperar corretamente o programa *assembly* original [4] e, para agravar ainda mais a situação, os programas de sistema que realizam essa tarefa simplesmente interrompem o processo de reversão do código sem reportar que erros foram detectados (i.e., eles falham silenciosamente). Justamente por isto, decidimos concentrar nossos esforços no sentido de obter o maior percentual possível de *bytes* corretamente recuperados: aquelas funções cuja recuperação for duvidosa, são invalidadas pelo nosso processo de reversão. Embora conservadora, esta estratégia permitiu recuperar mais do que 99% dos *bytes* que formam as instruções de cada programa do conjunto de *benchmarks* inteiros do SPEC2000.

Este relatório apresenta o *Disassembler*, uma ferramenta que realiza estaticamente a reversão do arquivo executável. A ferramenta utiliza dois algoritmos - um estático e outro dinâmico - para garantir sua correção. Também são realizados experimentos para obter as características arquiteturais de programas objeto da arquitetura CISC x86. Usamos como teste os programas inteiros do conjunto SPEC2000.

Este relatório está organizado em cinco Seções. A Seção 2 descreve o meio ambiente dos nossos experimentos e introduz relevantes conceitos relacionados com a reversão de código. A Seção 3 descreve os algoritmos utilizados na reversão de código. A Seção 4 apresenta os experimentos realizados e os resultados produzidos são analisados. Finalmente, as principais conclusões do trabalho estão condensadas na Seção 5.

2 Ambiente Experimental

O *Disassembler* teve como alvo a arquitetura Intel x86, por ser a arquitetura mais popular atualmente. Para o Sistema Operacional foi escolhido o GNU/Linux, por sua natureza *open source*, o que possibilita um maior controle sobre o ambiente sendo monitorado além de permitir modificações em seu *kernel* facilitando assim uma customização do sistema.

Nossos experimentos se basearam no subconjunto de *benchmarks* inteiros da suíte SPEC2000, largamente aceita pela comunidade de arquitetura de computadores como um conjunto para testes bastante representativa [19]. Os *benchmarks* foram compilados utilizando o compilador GNU/GCC, e o arquivo objeto foi gerado no formato ELF de 32 bits, padrão nos sistemas Linux.

2.1 O formato UNIX ELF

Um arquivo objeto pode se encontrar em diversos formatos, entre eles o os formatos `a.out`, `IBM/360`, `OMF`, `COFF` e o formato `PE - Portable Executable`, da Microsoft. Nossa plataforma se utiliza do formato UNIX `ELF - Executable and Linking Format` - de 32 bits, o qual descreveremos brevemente nesta Seção.

Um arquivo ELF pode ser basicamente de 3 tipos: relocável, objeto compartilhado ou executável. Um arquivo relocável é um arquivo não executável, próprio para uma posterior ligação. As informações de relocação e de símbolos estão presentes. Este tipo de arquivo serve com o entrada para um ligador, que produzirá um arquivo execu-

tável,já com todas as suas pendências com relação a relocações e símbolos resolvidas, com a exceção de alguns símbolos referentes a bibliotecas compartilhadas que só serão resolvidos em tempo de execução.

Objetos compartilhados são arquivos que possuem, além código executável, informações para o ligador. É um formato apropriado para as bibliotecas compartilhadas, amplamente utilizadas em sistemas Linux.

Um executável ELF pode ser visualizado logicamente de duas formas: como um conjunto de seções ou como um conjunto de segmentos, onde um segmento é composto por uma ou mais seções. Seções contém informações com semântica específica, como por exemplo uma tabela de símbolos, dados somente para leitura ou código de inicialização do ambiente. Um segmento normalmente é composto por um conjunto de seções, já preparadas para carga em memória. Compiladores e ligadores lidam com seções, enquanto carregadores lidam com segmentos.

Arquivos ELF têm sua estrutura descrita em um cabeçalho principal, o *ELF Header*. Neste cabeçalho podem ser encontradas as informações necessárias para uma correta manipulação do arquivo, como informações sobre o formato em que arquivo se encontra, ordem dos *bytes*, arquitetura alvo, número de seções e de segmentos, etc. As seções e segmentos também são descritos através de cabeçalhos especiais, que podem ser localizados a partir de informações no cabeçalho principal.

O *Disassembler* opera sobre arquivos executáveis estaticamente compilados, mas precisa de informações de relocação; desta forma o ligador precisa ser instruído a manter este tipo de informação no executável[1].

2.2 Seções de Código

Os arquivos ELF possuem código executável em 3 seções: `.init`, `.fini` e `.text`. Nas duas primeiras há menos do que uma dezena de instruções de máquina que fazem chamadas à funções contidas na seção de texto. Estas duas seções são invocadas quando o programa inicia e termina e, embora não sejam necessárias para programas escritos em C ou Fortran, são essenciais para os construtores estáticos da linguagem C++. A seção de texto, `.text`, contém o *core* executável, sendo assim o alvo efetivo da ferramenta.

Uma alternativa seria realizar um pré-processamento do arquivo executável, fazendo uma união das três seções. Desta forma, as instruções contidas nas seções `.init` e `.fini` [5] também seriam capturadas. O *Disassembler* não implementa esta estratégia de reescrita do executável (*binary rewriting*), pois foge ao seu escopo. Dinâmicamente, verificamos que estas seções contribuem somente com 21 instruções para o *instruction count*, em todos os programas.

2.3 Relocação

Outra seção importante no contexto da remontagem é a seção `.reloc`, que possui informações de relocação do arquivo, isto é, informações que identificam seqüências de bits do executável que correpondem à endereços do programa. Estas informações são necessárias caso o programa seja remapeado (por exemplo durante a ligação ou após a inserção/remoção de código) pois alguns endereços contidos nas instruções precisariam ser atualizados.

Em um arquivo ELF/x86, uma entrada desta tabela de relocações inclui um determinado deslocamento (*offset*) dentro de uma seção e o tipo de relocação (absoluta, relativa, etc.). Também pode estar codificada uma informação (*addend*) cuja interpretação depende do tipo de relocação. Na arquitetura x86 este *addend* não é necessário, pois está esta informação está presente diretamente na posição indicada como relocável.

2.3.1 Detecção de *Jump Tables*

Uma *jump table* é uma tabela de endereços usada por desvios indiretos, normalmente produzidas como resultado da estrutura de decisão *switch* da linguagem C. Cada endereço da tabela corresponde à um *case* da estrututura. Estas tabelas estão presentes em meio ao código e podem ser facilmente interpretadas como se fossem instruções por um *disassembler* desavisado.

Baseado nas informações de relocação, algumas *jump tables* podem ser detectadas [4] . Cada entrada da tabela, por definição, deve conter um endereço referente ao segmento de código, e desta forma deve possuir informações de relocação associadas à ela. Devido ao fato de que a arquitetura x86 só permite 2 endereços consecutivos dentro de uma mesma instrução [18], em qualquer seqüência de $E \geq 2$ endereços consecutivos marcados como

[1] Isto pode ser feito através da opção *-Wl,-q* durante a compilação

relocáveis, certamente os últimos $E-2$ pertecem à uma *jump table*.

Para determinar se os 2 primeiros endereços também pertecem à *jump table* ou à uma instrução, deve ser feita uma verificação junto à ultima instrução remontada e verificar quantos endereços consecutivos ela possuía no final, digamos, m. Então devem haver $2-m$ endereços entre o fim desta instrução e o início da *jump table* que a sucede, que na realidade são parte da *jump table*.

Soluções alternativas, que não se baseiam de informações de relocação, também podem ser empregadas quando o código fonte não está disponível [1][13]. Em essência, elas procuram identificar padrões no código que se referem ao uso de uma *jump table* e identificar seu endereço inicial e seu tamanho nas instruções correspondentes, identificando assim os *bytes* referentes à *jump table*.

2.4 Código Independente de Posição

Alguns compiladores são capazes de emitir código capaz de funcionar corretamente independentemente do endereço onde seja carregado. Este tipo de código é denominado PIC (*Position Independent Code*), e normalmente é encontrado em bibliotecas compartilhadas.

Para obter esta característica, são geradas *jump tables* também independenes de posição, isto é, tabelas que não armazenam endereços absolutos, mas deslocamentos. Para utilizar tais tabelas, uma instrução normalmente obtém o endereço da instrução que a sucede e efetua alguma aritmética utilizando-se dos *offsets* contidos na tabela para determinar seu endereço alvo.

Estas *jump tables* não possuem informações de relocação associadas à ela, já que não precisam ser modificadas pelo ligador e/ou carregador. A detecção destas tabelas é mais complicada, e, embora muitas vezes possível [1], o processo não é trivial.

2.5 Código Auto-Modificável

Um grande problema enfrentado pela Tradução e Interpretação Dinâmica é a presença de código auto-modificável (*self-modifying code*). O *software / hardware* deve tomar muito cuidado ao traduzir código auto-modificável, principalmente quando mecanismos tais como como cache de traduções (*Translation Cache*) [12], são utilizados.

Hsu [3] descreve um mecanismo implementado em sistemas Windows para detecção de áreas inicialmente não reservadas para código que posteriormente foram convertidas para tal fim. Para tal, são acessadas informações da LDT, ou *Local Descriptor Table*, em tempo de execução. A LDT é uma tabela de descritores de segmentos, onde cada entrada funciona como um seletor de segmento. Nela são armazenadas informações de proteção, o que impede um programa de executar algo dentro de seu segmento de dados, por exemplo.

3 Implementação

3.1 Visão Geral

O *Disassembler* opera sobre executáveis estaticamente ligados que retêm suas informações de relocação, o que, em sistemas GNU/Linux. O executável deve ser estaticamente ligado para que se possa incluir nas análises o código executado que corresponde às bibliotecas genéricas do sistema. As informações de relocação são utilizadas para a detecção de *jump tables* em meio ao fluxo de instruções.

Sua operação se dá basicamente em 3 etapas, sendo 2 estáticas e uma dinâmica, onde é feito um refinamento das informações obtidas estaticamente. A primeira etapa, (*Linear Sweep*), obtém instruções de forma seqüencial a partir do ponto de entrada do executável. A seguir é feita uma verificação recursiva das instruções obtidas, onde o fluxo de controle é respeitado.[2] Finalmente, devido aos desvios indiretos, cujos alvos não podem ser determinados estaticamente, é necessária uma última etapa, onde o programa é executado e monitorado, onde novos fluxos de controle são explorados. Esta metodologia é semelhante à adotada em [4], onde somente as 2 primeiras etapas são executadas.

Neste relatório, além da remontagem, procuramos também caracterizar os programas alvo, de forma que um refinamento dinâmico era indispensável, não apenas para garantir uma validação mais robusta da remontagem, mas também para a detecção de blocos básicos[3] formados por desvios indiretos.

[2] Desta forma, são remontados apenas *bytes* que realmente correspondem à instruções.

[3] Seqüência de instruções com apenas um ponto de entrada e um ponto de saída.

3.2 Preparando o *Disassembler*

O *Disassembler* mantém uma estrutura de dados que descreve cada *byte* da seção `.text`, aqui denominados *offsets*. À cada *offset* é associado um mapa de bits e um valor inteiro, de 8 *bytes*, utilizado durante a etapa dinâmica para armazenar o número de vezes que a instrução/bloco básico que inicia se naquele *byte* foi executado.

O mapa de bits tem por objetivo indicar o papel desempenhado pelo *offset* em vários contextos. Mais especificamente, um *offset* pode estar sinalizado como sendo:

1. O primeiro *byte* de uma relocação. Na arquitetura IA-32, as relocações se referem a grupos de 32 bits (4 *bytes*).
2. Um *byte* de alinhamento ou parte de uma *Jump Table*
3. O primeiro *byte* de uma instrução.
4. O *byte* pertencente à uma instrução. Se refere à todos os *bytes* que pertencem à uma instrução, exceto o primeiro.
5. O último *byte* de uma instrução.
6. O primeiro *byte* da primeira instrução de um bloco básico.
7. O último *byte* da última instrução de um bloco básico.
8. O primeiro *byte* da primeira instrução de uma função.
9. O último *byte* da última instrução de uma função.
10. Um *byte* que pertence à uma função cuja remontagem é duvidosa.

A primeira tarefa realizada pelo *Disassembler* é preencher esta estrutura com as informações de relocação (e a partir destas as *jump tables*) e delimitar as funções, para posterior utilização nas fases de remontagem e verificação. As relocações referentes à seção de texto estão na seção `.rel.text` e os pontos de entrada e tamanho das funções na seção `.symtab` (tabela de símbolos).

3.3 Reversão Linear

O algoritmo linear de remontagem original é o algoritmo mais simples possível para se efetuar a remontagem. O arquivo executável é analisado seqüencialmente a desde o primeiro *byte* até o último *byte* da seção de texto. É utilizado por ferramentas como o GNU *objdump* [11].

O *Disassembler* implementa uma versão modificada deste algoritmo, onde as informações de relocação são utilizadas para que *jump tables* não sejam interpretadas como instruções válidas, conforme descrito na Seção 2.3.1.

O algoritmo linear é apresentada a seguir:

```
1:  while (início da seção ≤ offset < fim da seção)
    do
2:    if (offset marcado como "dados") then
3:      Avançar até o final da jump table
4:    else if (offset marcado como relocável) then
5:      Marcar como "dados"os bytes entre offset e
        a jump table que segue
6:      Avançar até o final da jump table
7:    end if
8:    Remontar a instrução iniciando em offset
9:    Avançar offset para a próxima instrução
10: end while
```

3.4 Algoritmo Recursivo

O algoritmo linear é capaz de lidar com *jump tables* que possuam relocações associadas, mas as *jump tables* provenientes de PIC e *bytes* de alinhamento continuam sendo um problema. Normalmente, estes *bytes* são remontados como se fossem instruções, o que além de ser uma falha silenciosa[4], também pode causar a remontagem incorreta de instruções que sucedem estes *bytes* (embora o sincronismo seja novamente alcançado geralmente em uma ou duas instruções [14]).

O exemplo abaixo ilustra o problema causado por *bytes* de alinhamento inseridos em meio ao código. O trecho foi retirado da função `strrchr`, da biblioteca C padrão, e mostra como 3 *bytes* nulos, provavelmente inseridos pelo programador por razões de alinhamento, induzem o algoritmo linear à produzir resultados incorretos:

[4] dificilmente uma seqüência de *bytes* produz um código de operação inválido na arquitetura x86

```
0x8102195:   eb 3c              jmp 0x81021d3
0x8102197:   00 00              add %al,(%eax)
0x8102199:   00 83 ee 04 83 ee  add %al,
                                0xee8304ee(%ebx)
0x810219F:   04 83              add $0x83,%al
```

A seguir, a correta interpretação do código:

```
0x8102195:   eb 3c              jmp 0x81021d3
0x8102197:   00                 NULL
0x8102198:   00                 NULL
0x8102199:   00                 NULL
0x810219A:   83 ee 04 83 ee     lea 0xee8304ee
0x810219F:   04 83              add $0x83,%al
```

Uma alternativa seria efetuar a remontagem seguindo o fluxo de controle do programa. Esta abordagem não perminte que todas as intruções do programa sejam visitadas, devido aos desvios indiretos, mas pode servir como uma verificação do algoritmo de remontagem linear.

A verificação de pode ser feita da seguinte forma: ao remontar uma instrução em um determinado endereço, o algoritmo verifica se o algoritmo linear também desmontou uma instrução naquele endereço. Caso isto não ocorra, toda a função é marcada como problemática.

Embora conservador, em nenhum dos casos as funções invalidadas representaram mais de 0,5% do total de *bytes* do executável. O algoritmo recursivo é iniciado a partir do início de cada função, e pode ser descrito da seguinte forma:

```
1:  while (início da seção ≤ offset < fim da seção) do
2:    if (offset já foi visitado) then
3:      Retornar
4:    else
5:      Marcar offset como já visitado
6:      if (o algoritmo linear não desmontou uma
        instrução iniciando em offset ) then
7:        Marcar a função atual como problemática
8:      end if
9:      Remontar a instrução iniciando em offset
10:     if (a instrução é de desvio) then
11:       for (cada possível alvo) do
12:         if (alvo dentro da seção sendo remontada) then
13:           Reiniciar o algoritmo a partir dele
14:         end if
15:       end for
16:       Retornar
17:     end if
18:     Avançar offset para a próxima instrução
19:   end if
20: end while
```

3.5 Interpretação Binária

3.5.1 Motivação

Durante a execução dos algoritmos estáticos, são marcados blocos básicos à medida que instruções de desvio são encontradas. No algoritmo linear, embora os endereços alvo de desvios não possam ser marcados com segurança, endereços posteriores à instruções de desvio podem ser marcados como início de um novo bloco básico seguramente, pois caso a remontagem esteja incorreta, a função será posteriormente invalidada. Durante a verificação recursiva, os alvos podem ser marcados com segurança, mas, devido aos devios indiretos, nem todos os alvos podem ser determinados. Assim, uma última etapa é necessária.

3.5.2 A chamada de sistema `Ptrace`

O Sistema Operacional Linux inclui uma *syscall* denominada `ptrace`, que foi usada para monitorar e controlar a execução de binários. Em outras palavras, usamos esta função do Linux para implementar a Interpretação Binária de código x86 num processador x86.

Ativando `ptrace`, um processo *supervisor* pode se *anexar* à um outro processo *alvo* e monitorar sua execução, recebendo o controle à cada instrução executada [5]. O supervisor pode então manipular a imagem do processo alvo, através da leitura e/ou escrita em qualquer *byte* no seu espaço de endereçamento ou em seus registradores.

O *Disassembler* utiliza esta Interpretação para determinar as instruções que foram realmente executadas e também para capturar alvos de desvios indiretos. Neste contexto, o papel do supervisor é o de manter os contadores de execução de cada instrução, além de verificar se a remontagem naquele endereço coincide com as informações estáticas obtidas nas fases anteriores.

[5] Na realidade, o controle também pode ser devolvido ao processo supervisor em situações especiais, tais como quando o processo alvo realiza uma *syscall*.

3.5.3 Monitoração Dinâmica

Através da chamada de sistema `ptrace`, a execução do programa é monitorada para se obter os endereços alvo de instruções de desvio indireto. Novos blocos básicos são determinados, como conseqüência direta.

O algoritmo utilizado é bastante similar ao algoritmo recursivo, mas é seguido somente o fluxo de controle que o programa tomou. Nesta fase, também é feita a verificação das informações de remontagem obtidas estaticamente.

A seguir apresentamos uma descrição simplificada das ações realizadas pelo algoritmo ao receber o controle:

1: *offset* = PC - endereço base da seçao sendo remontada
2: **if** ($0 \leq$ *offset* $<$ tamanho da seção) **then**
3: **if** (os algoritmos estáticos não remontaram uma instrução iniciando em *offset*) **then**
4: Marcar a função atual como problemática
5: **end if**
6: Atualizar o contador de execuções da instrução
7: **if** (PC $\neq$ PCesperado) **then**
8: Marcar o início de um novo bloco básico
9: **end if**
10: Remontar a instrução apontada pelo PC
11: PCesperado = endereco *fall-through*
12: **if** (a instrução é de desvio) **then**
13: Marcar os alvos como novos blocos básicos
14: **end if**
15: **end if**

4 Experimentos

Nossos experimentos foram realizados em uma máquina Intel Pentium IV, com 512 MB de memória, utilizando Red Hat Linux 7.3, e a suíte de *benchmarks* utilizada foi o SPECint-2000, compilados com `-tune=base` e com as otimizações `-O2 -funroll-loops`, exceto *eon* e *gap*, compilados com `-O0 -funroll-loops` e `-O3 -funroll-loops`, respectivamente. Também durante a compilação, o compilador foi instruído a manter as informações de relocação e, como pré-requisito, gerar executáveis ligados estaticamente. O compilador utilizado foi o *gcc* 2.96.

Inicialmente foi efetuada a remontagem linear, verificada estaticamente a seguir. O refinamento dinâmico, inicialmente apenas com o objetivo de verificação e detecção de novos blocos básicos, foi realizado utilizando o conjunto de entradas reduzido [8] para os *benchmarks*.

A seguir, a massa de dados *test*[6] foi utilizada para se obter estatísticas dinâmicas, entre elas o total de instruções dinâmicas e a quantidade de vezes que cada instrução/blocos básico que era executado.[7] Os resultados são apresentados em seguida.

4.1 Resultados

Algumas características dos objetos x86 puderam ser determinadas durante o processo de refinamento dinâmico. A Tabela 1 mostra o *instruction count*, o total de instruções e de blocos básicos para cada programa do conjunto de *benchmarks*.

4.2 Uso de Instruções e Blocos Básicos

As Tabelas 2 e 3 revelam uma característica dinâmica muito importante dos programas: a baixa utilização de instruções e blocos básicos. Poucas instruções são responsáveis por todo o *instruction count*. No caso específico das instruções, o percentual de instruções que são executadas ao menos uma vez é muito baixo, principalmente se levado em consideração que o compilador foi instruído a desenrolar os os laços.

O percentual de utilização das instruções não foi uniforme, com valores acentuadamente baixos (7.7%) em programas como *gzip* e valores maiores (35%) em outros, como o *gcc*. No caso dos blocos básicos, o percentual de utilização se mostrou semelhante ao das instruções, embora um pouco menor, com um máximo de 32%, no *gcc*.

Estes percentuais são muito baixos, o que certamente produz um efeito negativo nas caches de instruções e/ou nas *trace caches*, já que muito do espaço de armazenamento é gasto com instruções que nunca chegarão a ser executadas de fato. Estas características justificam esforços de otimização do *layout* do código destes objetos, sendo o trabalho de Pettis e Hansen [15] um dos mais importantes, e mais recentente, o trabalho de Schwarz [5], que aplica várias técnicas de otimização em objetos da arquitetura x86.

Um fator importante que contribui para esta baixa taxa de aproveitamento de instruções é o fato

[6] Para o programa *eon* foi utilizada a entrada `kajiya`
[7] Para o programa *perlbmk* foi utilizado conjunto *reference*, entrada `makerand`

Tabela 1: Características dos Programas

Programa	Instr. Count	Instruções	Blocos Básicos
gzip	2,453,607,555	107,515	28,780
gcc	1,491,388,864	423,765	118,357
vpr	1,565,238,393	125,610	31,924
mcf	193,182,166	89,665	24,372
eon	2,226,460,008	247,528	50,992
gap	872,946,094	242,666	64,416
bzip2	8,283,448,196	100,964	26,872
twolf	227,743,582	144,573	36,575
crafty	5,349,223,167	135,691	33,819
vortex	8,715,244,665	213,637	52,756
parser	3,314,599,773	112,403	30,919
perlbmk	1,342,912,985	220,868	60,522

Tabela 2: Instruções

Programa	Instruções	Executadas	Exec (%)
gzip	107,515	8,285	7.71
gcc	423,765	150,270	35.46
vpr	125,610	18,416	14.66
mcf	89,665	7,503	8.37
eon	247,528	54,976	22.21
gap	242,666	42,981	17.71
bzip2	100,964	8,571	8.49
twolf	144,573	28,589	19.77
crafty	135,691	36,754	27.09
vortex	213,637	67,718	31.70
parser	112,403	23,129	20.58
perlbmk	220,868	19443	8.8

de que em executáveis compilados estaticamente, as bibliotecas são incorporadas por completo ao arquivo objeto, não somente as funções realmente referenciadas. O efeito da ligação estática e dinâmica sobre o total de instruções intocadas é um dos pontos sob nossa investigação.

4.3 Tamanho dos Blocos Básicos

O tamanho dos blocos básicos também foi um ponto que mereceu destaque. Com exceção do programa *mcf*, todos os programas apresentaram um aumento no tamanho médio dos blocos básicos, tanto em *bytes* como em número de instruções, sendo que no segundo caso, mesmo o *mcf* apresentou um pequeno aumento, já que o tamanho dinâmico de suas instruções é menor.

Estaticamente, o número de instruções por bloco básico variou entre 3.58 (*gcc*) e 4.8 (*eon*), uma variação de 34%, assim como o total de *bytes* por bloco básico, que variou entre 11.26 e 15.35 para os mesmos programas, ou seja, 36%. Já dinâmicamente, o tamanho médio dos blocos básicos aumentou, tanto em número de instruções quanto em número de *bytes*. O programa *gcc* apresentou os menores blocos básicos dinâmicos, com 4.02 instruções em média, e o programa *mcf* apresentou os menores blocos básicos em termos de tamanho em *bytes*, 10.78. O programa *bzip2* apresentou os maiores blocos básicos, com 7.14 instruções e 29.83 *bytes* em média. A Tabela 4 ilustra as características dos blocos básicos.

Tabela 3: Blocos Básicos

Programa	Blocos Básicos	Executados	Exec (%)
gzip	28,780	1852	6.44
gcc	118,357	38,668	32.67
vpr	31,924	3,888	12.18
mcf	24,372	1,707	7.00
eon	50,992	8,686	17.03
gap	64,416	9,904	15.38
bzip2	26,872	1,919	7.14
twolf	36,575	6,077	16.62
crafty	33,819	6,249	18.48
vortex	52,756	14,037	26.61
parser	30,919	5,954	19.26
perlbmk	60,522	4,574	7.56

Tabela 4: Tamanho Médio dos Blocos Básicos

Programa	Bytes Estáticos	Instr. Estáticas	Bytes Dinâmicos	Instr. Dinâmicas
gzip	12.352	3.736	17.624	4.916
gcc	11.261	3.580	12.314	4.017
vpr	12.686	3.935	15.520	5.151
mcf	11.941	3.679	10.782	4.091
eon	15.351	4.854	17.415	6.602
gap	11.819	3.767	12.800	4.398
bzip2	12.443	3.757	29.826	7.136
twolf	13.134	3.953	14.561	5.085
crafty	13.796	4.012	23.817	7.016
vortex	12.558	4.050	12.992	4.878
parser	11.620	3.635	11.403	4.383
perlbmk	11.558	3.649	14.611	4.978

4.3.1 Fragmentos

Um conceito importante na Ciência da Computação é o de um *fragmento*. Um fragmento pode ser definido por uma seqüência de instruções onde uma instrução de desvio aparece somente no fim [17]. Com efeito, um fragmento pode ser intepretado como um conjunto de blocos básicos consecutivos, onde somente o último bloco básico termina com uma instrução de desvio.

Desta forma, um fragmento é um grande bloco com vários pontos de entrada, e sua deteção é uma tarefa semelhante à quantificar os blocos básicos que não terminam com uma instrução de desvio. Uma conseqüência direta da existência de fragmentos longos (ie, com muitos blocos básicos) é a deterioração de mecanismos como a *trace cache*, já que não possuem mecanismos de aproveitamento de *traces* parciais. Desta forma, estamos investigando o ofeito dos fragmentos sobre esta redundância.

5 Conclusões

Neste trabalho descrevemos a implementação de um *Disassembler* de código Intel x86 gerado para o Sistema Operacional Linux. O processo de conversão de código emprega dois algoritmos estáticos e em seguida usa a interpretação binária (algoritmo dinâmico) para validar as instruções detectadas anteriormente. Essa detecção dinâmica complementa a tarefa dos algoritmos estáticos pois ela permite recuperar novas instruções. Elas são aquelas alvejadas por transferências de controle indiretas cujos endereços alvos não podem ser determinados estaticamente. Nosso processo de reversão de código é capaz de lidar com tabelas do tipo *jump tables* que aparecem misturadas com as instruções do programa binário, desde que as respectivas informações de relocação estejam disponíveis. Os dois algoritmos usados na recuperação estática foram: os algoritmos linear e recursivo.

Além de validar e complementar a recuperação estática, nosso esquema de interpretação binária determinou importantes características arquiteturais de máquinas CISC. Os programas inteiros do conjunto SPEC2000, foram interpretados integralmente e para cada programa determinamos a percentagem de instruções que foram utilizadas pelo menos uma vez. Verificamos que o percentual de instruções e blocos básicos executados pelo menos uma vez é muito modesto, o que é um fator importante a ser considerado quando do projeto de novas arquiteturas. Constantamos também que os blocos básicos, em geral, são formados por 3 ou 4 instruções, assim como o tamanho médio em *bytes* das instruções.

Como trabalhos futuros, pretendemos investigar o efeito da ligação dinâmica das funções das bibliotecas compartilhadas dinâmicas no número de instruções que permanecem intocadas ao longo da execução. Acreditamos também que é necessário determinar as características dos fragmentos do código mais profundamente.

Referências

[1] C. Cifuentes, K. J. Gough, "Decompilation of Binary Programs", *Software-Practice and Experience* 25(9), 1995.

[2] D. Ung, and C. Cifuentes, "Machine-Adaptable Dynamic Binary Translation,"*Proceedings of the ACM Workshop on Dynamic Optimization*, Dynamo '00, 2000.

[3] Liangchuan Hsu, "A Robust Foundation for Binary Translation of x86 Code,"*Ph.D. Thesis*, Departament of Computer Science, University of Illnois at Urbana-Champaign, 1997.

[4] B. Schwarz, S. Debray and G. Andrews, "Disassembly of Executable Code Revisited,"*Proceedings of 2002 Working Conference on Reverse Engineering*, 2002.

[5] B Schwarz, "Post Link-Time Optimization on the Intel IA-32 Architecture,"*Honors Thesis*, University of Arizona, 2002.

[6] The Bastard Disassembly Environment, `http://bastard.sourceforge.net/libdisasm.html`.

[7] Standard Performance Evaluation Corporation, CPU2000, `http://www.spec.org/osg/cpu2000`.

[8] Reduced input sets for SPEC CPU2000, `http://www.spec.org/osg/cpu2000/research/umn/`.

[9] M. Probst. "Fast machine-adaptable dynamic binary translation,"*Proceedings of the Workshop on Binary Translation*, Barcelona, Spain, 2001.

[10] J. R. Levine, *Linkers and Loaders,* Morgan Kauffman, 2000.

[11] GNU Project - Free Software Foundation, `objdump`, *GNU Manuals Online,* `http://www.gnu.org/manual/binutils-2.10.1/html_chapter/binutils_4.html` .

[12] A. Klaiber, "The Technology Behind Crusoe Processors", Transmeta Corporation, 2000.

[13] C. Cifuentes e M. Van Emmerick, "Recovery of Jump Table Case Statements from Binary Code,"*Proceedings of the International Workshop on Program Comprehension*, 1999.

[14] C. Linn, S. Debray."Obfuscation of Executable Code to Improve Resistance to Static Disassembly,"*To appear in Proceedings of 10th ACM Conference on Computer and Communications Security*, CCS '03, 2003.

[15] K. Pettis e R. C. Hansen. "Profile Guided Code Positioning,"*Proceedings of the Conference on Programming Language Design and Implementation*, 1990.

[16] A. Ramirez, L. A. Barroso, K. Guarachorloo, R. Cohn, J. Larriba-Pey, P.G. Lowney e M. Valero. "Code layout Optimization for Transaction Processing Workloads,"*Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA'01, 2001.

[17] K. Scott, N. Kumart, S. Velusamy, B. Childers, J.W. Davidson e M. L. Soffa. "Retargetable Reconfigurable Software Dynamic Translation,"*Proceedings of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'03, 2003.

[18] IA-32 Intel Architecture Software Developer's Manual, Vol 2. Intel Corporation.

[19] D. Citron. "MisSPECulation: partial and misleading use of SPEC CPU2000 in Computer Architecture Ponferences", *Proceedings of the 30th Annual international symposium on Computer architecture*, ISCA'03, 2003.