Uma Infra-estrutura Modular e Extensível para Otimização de Binários

Relatório de Andamento

Ricardo Nabinger Sanchez

I. 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 linguagem de máquina. Para contornar estas limitações, pesquisadores introduziram linguagens de mais alto nível e consequentemente, os primeiros compiladores capazes de realizar a "programação automática" [1].

Parte do sucesso das linguagens de programação de alto nível se deve à qualidade e ao volume 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 objetivam um aproveitamento cada vez maior dos recursos computacionais disponíveis a partir do refinamento e desenvolvimento de novas técnicas de otimização. Parte desse 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 dessas interações [2].

Compiladores como o GNU Compiler Collection (GCC) [3] implementam diversas técnicas de otimização para as arquiteturas suportadas, tanto genéricas quanto específicas[1]. O GCC organiza a compilação em um conjunto de passos pré-determinados. A cada passo é realizada uma técnica de otimização, sendo que a ordem de aplicação das técnicas é rígida, isto é, não pode ser alterada pelo usuário. Esta rigidez é desvantajosa, especialmente quando ocorre uma interação negativa entre duas ou mais técnicas em função da ordem de aplicação das mesmas.

Trabalhos recentemente publicados propõem diversos métodos, ferramentas e ambientes relevantes para esta área, tais como: *(i)* coordenação da ordem de aplicação das técnicas de otimização [2], *(ii)* aplicação exaustiva de técnicas de otimização[4], [5], *(iii) frameworks* para implementação de novas técnicas de otimização [6], [7], *(iv)* otimizadores em nível de linguagem de montagem [8] e *(v)* reengenharia de código atuando diretamente nos binários executáveis [9].

[1]Técnicas genéricas de otimização não consideram os detalhes da arquitetura-alvo por atuarem no nível da linguagem de programação, enquanto que as específicas procuram explorar os recursos disponíveis da arquitetura-alvo.

Este trabalho propõe uma infra-estrutura modular para a desmontagem de binários e reengenharia de código, onde cada módulo implementa uma diferente técnica de otimização. A infra-estrutura é naturalmente extensível para a desmontagem e reengenharia de diversas arquiteturas-alvo. Porém, como estudo de caso, foi escolhida a arquitetura MIPS por ser RISC[2] em conjunto com o formato de binários executáveis ELF para sistemas UNIX, padrão *de facto* nestes sistemas.

O restante deste relatório é dividido em cinco seções. A Seção II apresenta diversos trabalhos relacionados que utilizam as técnicas consideradas estado-da-arte em desmontagem, reengenharia e otimização de código, seguida pela Seção IV que descreve as técnicas adotadas no trabalho proposto. O problema e a abordagem adotada para tratá-lo são descritos na Seção III. Resultados obtidos até o momento são apresentados na Seção V, seguida pelas conclusões e trabalhos futuros na Seção VI.

II. Estado-da-arte

O trabalho proposto demanda a investigação de técnicas consideradas estado-da-arte para três áreas distintas: *desmontagem de binários*, *reengenharia de código*, e *técnicas de otimização*. A primeira é utilizada para decodificar as instruções de máquina presentes no programa a ser modificado; a segunda permite que a estrutura de controle descoberta na fase de decodificação seja alterada preservando sua semântica; e finalmente a terceira procura formas de reduzir a quantidade de recursos computacionais e/ou o tempo de computação, necessários para realizar determinadas ações.

Até o momento da escrita deste relatório, as áreas de reengenharia de código e técnicas de otimização estão apenas sendo investigadas. Portanto, apenas a desmontagem de binários será detalhada, na Seção a seguir.

A. Desmontagem de Binários

Um arquivo que contenha instruções executáveis pode ter seu conteúdo decodificado tanto de forma *estática* quanto *dinâmica*. Diz-se que a desmontagem é estática quando o conteúdo do arquivo é decodificado antes de sua execução em um ambiente, que não necessita recriar um ambiente oferecido pelo sistema operacional ou arquitetura subjacente. A desmontagem dinâmica, por outro lado, transcorre no ambiente real

[2]RISC, computador com um conjunto de instruções reduzido, em Inglês, *Reduced Instruction Set Computer*.

de execução do código, decodificando as instruções momentos antes de sua execução. Considerações aprofundadas sobre as técnicas de desmontagem podem ser encontradas em [10], [11].

1) Desmontagem Estática: Na literatura encontram-se basicamente três técnicas de desmontagem estática: varredura linear, caminhamento recursivo e desmontagem especulativa.

A mais simples é a *varredura linear*, usada por ferramentas simples que assumem que a seção do arquivo executável sendo desmontada contém apenas instruções ([11]). Por ser muito simples, esta técnica requer poucos recursos computacionais, porém é bastante imprecisa [12], [11], e portanto não será adotada no trabalho proposto.

O *caminhamento recursivo* parte da premissa que instruções e dados podem estar misturados ao longo da seção, e sendo assim a desmontagem deve constantemente avaliar o que encontra para distinguir entre dados e instruções [13], [14], [15], [16], [12], [17]. Sendo assim, a desmontagem ocorre conforme a semântica das instruções, isto é, a desmontagem tem início no ponto de entrada[3] do programa e segue conforme o fluxo de controle das instruções—daí o nome da técnica. Uma extensão a esta técnica será adotada neste trabalho, e detalhada na Seção IV-A.

Uma característica do caminhamento recursivo é o aparecimento de *sombras*— regiões que não foram analisadas pelo desmontador por não haver referências estáticas que o levem até estas regiões. O problema está em remover (de forma indesejada) estas regiões do binário que será gerado após o processamento do original, resultando em um binário incompleto (com semântica diferente) ou até mesmo inválido.

Alguns autores propõem extensões ao caminhamento recursivo [12], [11], adotando heurísticas para decidir se determinadas regiões não analisadas possuem uma probabilidade suficiente de realmente representarem instruções do programa que está sendo desmontado, e portanto necessitando ser analisadas. Esta técnica é conhecida como *desmontagem especulativa*, e possui diversas variações, em especial quanto às heurísticas adotadas para decidir se uma determinada região é composta por instruções, e geralmente levam em consideração aspectos específicos de compiladores e arquiteturas [12]. Devido à natureza dos problemas que este trabalho se propõe a tratar (ver Seção III), esta técnica não será adotada, embora seja relacionada por atuar sobre as sombras que por ventura existam nos binários processados.

2) Desmontagem Dinâmica: A desmontagem dinâmica manipula instruções de binários em tempo de execução [18], [9], [19]. As técnicas usadas são as mesmas da desmontagem estática descritas na Seção anterior, acrescidas de maior complexidade para preservar a semântica de execução do binário.

Diversos trabalhos existentes na literatura propõem técnicas de otimização dinâmica e tradução binária (e não raro ambas). Otimizações dinâmicas precisam necessariamente ser feitas durante a execução de um binário em um ambiente de otimização como os trabalhos propostos em [18], [9]. Estes trabalhos atuam não apenas no caminho crítico de execução dos binários (também conhecido como *hot-spots*), mas também nas regiões que o ambiente virtual considerar vantajoso. Em especial, [18] adota uma técnica de desmontagem *progressiva*, onde um bloco de instruções é incrementalmente desmontado conforme sua utilização[4]. Algumas das técnicas de otimização descritas em [18] poderão ser adotadas neste trabalho, como por exemplo a redução de força de instruções (*strength reduction*, em Inglês).

Em [19] os autores propõem uma arquitetura para tradução binária dinâmica, isto é, realizada momentos antes das instruções do programa serem executadas. A tradução binária consiste em converter um conjunto E de instruções para um conjunto S, preservando a semântica da operação, pois nem sempre existe um mapeamento 1:1 entre as operações realizadas pelas instruções dos conjuntos E e S. Especificamente para fins de validação e análise de desempenho, [19] realiza tradução binária entre os conjuntos de instruções dos processadores Intel Pentium MMX e Sun UltraSparc II, tanto entre ambos os processadores ($E_{Intel} \rightarrow S_{Sun}$ e $E_{Sun} \rightarrow S_{Intel}$) quanto para o mesmo processador ($E_{Intel} \rightarrow S_{Intel}$ e $E_{Sun} \rightarrow S_{Sun}$), para fins de validação e análise de desempenho. O trabalho aqui proposto possibilita a tradução binária de forma estática (sem a presença do ambiente de execução), embora este não seja o seu foco inicial.

3) Desmontagem Híbrida: Alguns trabalhos propõem a união das técnicas de desmontagem de forma a gerar uma técnica híbrida [20], [21], tendo em vista que estas possuem características complementares [11].

Em [20] os autores propõem uma infra-estrutura para instrumentação de binários, empregando a desmontagem estática (usando caminhamento recursivo, detalhado na Seção II-A.1) em um primeiro momento. Após a desmontagem inicial, o binário é executado em um ambiente oferecido pela infra-estrutura para que as sombras possam ser desmontadas e instrumentadas, desta vez empregando a desmontagem dinâmica e também a desmontagem especulativa. Com isso, os autores alegam obter *índices de cobertura*[5] maiores dos que os obtidos com ferramentas comerciais para desmontagem, utilizando binários comerciais que não possuem informações simbólicas que poderiam auxiliar na desmontagem.

Em [21] os autores descrevem um tradutor binário que também realiza otimizações dinâmicas após a tradução, em especial técnicas que visam tirar maior proveito da localidade de código, como a redisposição e consolidação de blocos-básicos. A infra-estrutura implementa outras técnicas de otimização dinâmica, e por instrumentar o código sendo otimizado, pode atuar no caminho crítico do programa, obtendo rapidamente resultados significativos com um menor custo computacional.

III. DESCRIÇÃO DO PROBLEMA

A principal motivação para manipular diretamente código binário é a portabilidade entre linguagens de programação de alto-nível, isto é, não importará qual tenha sido a linguagem

[3] *Ponto de entrada* de um programa é o endereço virtual ou real de memória onde o processador iniciará a execução do mesmo.

[4] Esta utilização pode ser medida com *profilers* executando sobre a arquitetura-alvo.

[5] Índice de cobertura no contexto de desmontagem é a taxa de instruções decodificadas pelo desmontador em comparação com as efetivamente geradas pelo compilador.

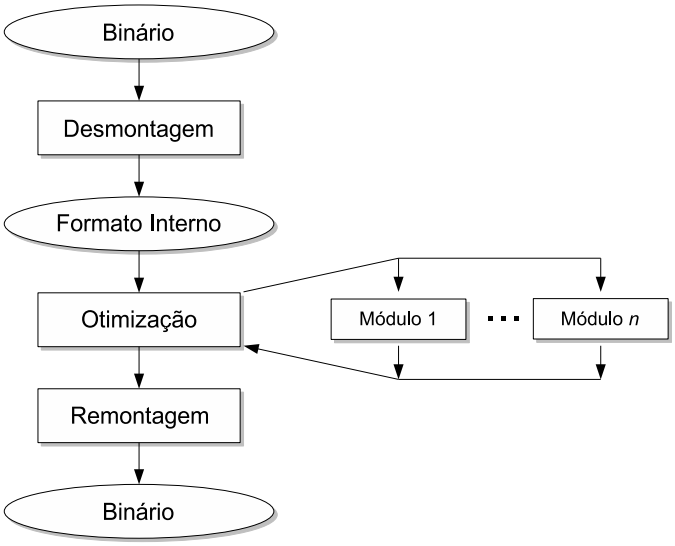

Fig. 1. Arquitetura da infra-estrutura proposta.

adotada para implementar determinado programa, desde que ele possa ser compilado em instruções de máquina. Além disto, a otimização de binários pode ser mais eficaz por tratar código de nível mais próximo ao da arquitetura alvo.

O trabalho proposto visa oferecer uma infra-estrutura de *software* para coordenar a aplicação de módulos que implementem técnicas de otimização para binários executáveis. A infra-estrutura fica responsável por desmontar, estruturar a representação intermediária, e montar um novo binário a partir da estrutura (possivelmente) modificada. Os módulos implementam técnicas de otimização, manipulando a estrutura interna que representa o binário desmontado (um grafo, detalhado na Seção IV-C).

A extensibilidade da infra-estrutura é conferida pelos próprios módulos, pois a infra-estrutura não assume ou exige que os módulos implementem apenas técnicas de otimização e tampouco que estas sejam direcionadas a uma arquitetura alvo específica, embora técnicas de otimização para arquiteturas RISC serão as estudadas e desenvolvidas no trabalho proposto. O papel dos módulos na infra-estrutura não se limita a técnicas de otimização, pois a estrutura para carregar dados do binário executável, decodificar e gerar um novo arquivo também é modular. Para incluir suporte a arquiteturas ou formatos de binários executáveis adicionais, implementa-se as funcionalidades a partir dos *frameworks* definidos para estes módulos, e indica-se para a infra-estrutura que tipo de funcionalidades eles oferecem, para que durante a execução ela verifique se o módulo é adequado para tratar o binário a ser analisado.

A Figura 1 ilustra o fluxo de atividades que a infra-estrutura desempenhará. As *elipses* representam dados, enquanto que os *retângulos* representam operações. A infra-estrutura recebe o binário executável como entrada, desmonta-o, e gera uma estrutura em forma de grafo (detalhada na Seção IV-C). Este grafo é passado para cada um dos módulos, que eventualmente o modificam. Depois de aplicar o n-ésimo módulo, a infra-estrutura gera um novo binário executável a partir do grafo, possivelmente modificado.

IV. TÉCNICAS ADOTADAS

Para a desmontagem de binários executáveis, este trabalho adota e estende a técnica de *caminhamento recursivo*. O resultado da desmontagem é armazenado em um grafo de controle de fluxo. A técnica de desmontagem adotada está descrita na Seção IV-A, seguida pelo detalhamento da divisão de blocos-básicos na Seção IV-B, e finalmente pelas estruturas de dados mais relevantes, detalhadas na Seção IV-C.

A. Desmontador

O desmontador estende o algoritmo tradicional de caminhamento recursivo para decodificar instruções presentes nas seções apropriadas do arquivo executável, buscando formar blocos-básicos para re-estruturar logicamente o controle de fluxo do binário executável. Arquivos ELF consistem de um cabeçalho principal no início do arquivo, um ou mais cabeçalhos para os segmentos, e os segmentos propriamente ditos, que contêm os dados do arquivo [22]. Uma seção em um arquivo ELF é considerada executável se ela possui o atributo `SHF_EXECINSTR` ativado no campo apropriado de seu respectivo cabeçalho de seção.

Um *bloco-básico* é um segmento de instruções, sendo que dentro dele não existem instruções que alteram o fluxo de execução (instruções para desvios condicionais e incondicionais) [23].

A extensão ao algoritmo de caminhamento recursivo consiste em uma premissa adicional para tratar o eventual aparecimento de sombras durante a desmontagem, isto é, regiões que em princípio não fariam parte de nenhum bloco-básico. Em um primeiro momento, o algoritmo assume que cada seção executável do arquivo representa um bloco-básico, e que a desmontagem começará pelo ponto de entrada do programa. É importante ressaltar que as características que definem um bloco-básico são inicialmente violadas.

O desmontador então analisa cada instrução para determinar o endereço da próxima instrução. Quando se depara com uma instrução que causa desvio[6] do fluxo de controle, ele verifica se o endereço virtual do destino pode ser determinado estaticamente. Se possível, o desmontador divide o bloco-básico e, recursivamente, explora os destinos dos desvios recém encontrados. Ao final da recursão, o desmontador terá analisado todos os blocos-básicos que podem ser analisados estaticamente, de forma conservadora[7], sendo que as regiões não analisadas (sombras) serão preservadas pelos blocos gerados indiretamente pelas divisões.

B. Divisão de Blocos-básicos

A divisão de blocos-básicos é uma característica importante do algoritmo adotado, pois é esta divisão que permitirá a representação apropriada do fluxo de controle de um programa.

A Figura 2 ilustra a divisão de blocos-básicos. Tem-se inicialmente em (a) dois blocos-básicos, com o ponto de

[6]Um desvio pode ser condicional (onde há dois possíveis destinos), incondicional (apenas um destino possível), ou uma interrupções de *software* (eventualmente a execução é retomada na instrução que segue a interrupção).

[7]Forma conservadora não adota técnicas especulativas de desmontagem.

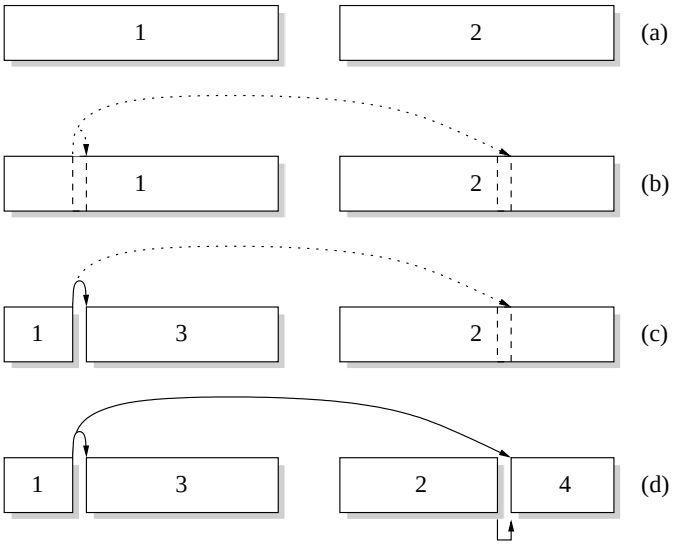

Fig. 2. Etapas para a divisão de blocos-básicos

entrada no bloco B_1, e ambos referenciados pela estrutura que mapeia os espaços de endereçamento. Em (b) o desmontador localiza uma instrução de desvio, que causará a divisão dos blocos envolvidos, começando pelo bloco onde a instrução se encontra, como ilustrado em (c), onde o bloco B_1 é dividido nos blocos B_1 e B_3. O exemplo ilustra um *desvio condicional*, cuja característica é a presença de dois destinos: o ramo *tomado*, representado pelo salto longo, e o ramo *não tomado*, que aponta para a próxima instrução a ser executada[8]. Finalmente em (d) o bloco B_2 é dividido, dando origem ao bloco B_4, e a referência $B_1 \rightarrow B_4$ é definida, representando o ramo *tomado*. É importante ressaltar que a relação $B_2 \rightarrow B_4$ não é equivalente a $B_1 \rightarrow B_3$ ou $B_1 \rightarrow B_4$, esta relação só existe para representar os casos onde, logicamente, existe mais de um ponto de entrada, como pode ocorrer na implementação de comandos condicionais da linguagem C do tipo `switch` (*expr*) `{...}`.

C. Representação Interna

Neste trabalho, todos os blocos-básicos são referenciados por uma estrutura que representa o espaço de endereçamento virtual do binário executável. Sua importância fica evidente quando se faz necessário localizar o bloco-básico ao qual pertence um determinado endereço. Isto acontece quando o desmontador precisa localizar a instrução (e/ou seu endereço virtual) referenciada por uma instrução de desvio.

Os blocos-básicos também são referenciados por uma estrutura em forma de grafo direcionado, que representa o grafo de controle de fluxo (CFG, *Control Flow Graph* em Inglês). No contexto deste trabalho, os vértices (representados pelos blocos-básicos) podem ser "apontados" por múltiplas arestas, mas cada vértice pode ter até duas arestas partindo de si. Além disso, está prevista a *concatenação* de vértices utilizando uma aresta especial, que serve apenas para garantir que a premissa de que um bloco-básico possui apenas uma entrada e uma saída não seja violada. Quando esta aresta especial é utilizada, as outras duas (reservadas para os ramos de desvio *tomado* e *não tomado*) não apontam para blocos-básicos, pois a última instrução do bloco não é um desvio.

Internamente aos blocos básicos, as instruções são representadas através de uma lista duplamente encadeada. Dentre as principais razões para a adoção de uma lista encadeada, pode-se citar a inserção, remoção e relocação das instruções dentro de um bloco-básico. Os binários analisados até o momento apresentaram blocos-básicos com poucas instruções, sendo que os maiores continham menos de 30 instruções.

V. Resultados Parciais

A infra-estrutura aqui proposta está sendo implementada na linguagem C, em ambiente UNIX (sistema operacional FreeBSD, mais especificamente). A infra-estrutura recebe como entrada um binário executável ELF32/UNIX compilado para arquitetura MIPS. O compilador usado é o MIPS SDE Lite 6.05.00[9] (versão para sistemas GNU/Linux). Parte das funcionalidades previstas e implementadas estão descritas nos itens que seguem:

- Localizar e carregar seções executáveis de binários ELF32/UNIX para processadores MIPS, identificando o conjunto de instruções MIPS32 [24].
- Se necessário, a plataforma realiza a conversão entre diferentes representações de bytes (*endianess*, podendo ser *little-endian*, quando o byte mais significativo de uma palavra está no endereço de memória *maior* do que os bytes restantes da palavra; ou *big-endian*, quando o byte mais significativo está em um endereço menor que os demais) de forma automática.
- Desmontagem das instruções usando a técnica descrita na Seção IV-A.
- Descoberta dos limites dos blocos-básicos, realizando a divisão dos mesmos conforme detalhado na Seção IV-B.
- Manutenção do espaço de endereçamento virtual apresentado na Seção IV-C, necessário para a correta localização de instruções em seus respectivos blocos-básicos.

A Figura 3 ilustra em a saída gerada pelo desmontador, enquanto que a Figura 4 ilustra a gerada pela ferramenta GNU `objdump`. A saída do desmontador implementado é bastante primitiva por ter sido recentemente gerada, e como é suficiente para a depuração, não foi aprimorada. A última instrução apresentada na Figura 3 é um exemplo de instrução MIPS que é decodificada em duas etapas: a primeira identifica que trata-se de um agrupamento de instruções, e na segunda etapa a instrução exata é determinada.

Dentre as dificuldades encontradas, as maiores foram sem dúvida as limitações intrínsecas da desmontagem estática, citadas ao longo deste relatório. A extensão proposta para o algoritmo de desmontagem (Seção IV-A) visa amenizar esta problemática, mas não a resolve completamente, pois mesmo assim ainda existem diversas situações onde um bloco de instruções não é referenciado estaticamente. Já se sabe que a infra-estrutura não será capaz de lidar adequadamente

[8]Na arquitetura MIPS, o ramo *não tomado* aponta na verdade para a segunda instrução contando a partir do desvio.

[9]Disponível em `http://www.mips.com/content/Products/SoftwareTools/SDE_Lite/content_html`

```
Machine type: 8 (0x08)
Using ‘MIPS32’ ELF module
27bdffe0 addiu (9)
afb10014 sw (43)
3c118002 lui (15)
922201b8 lbu (36)
afbf0018 sw (43)
14400012 bne (5)
afb00010 sw (43)
10000004 beq (4)
3c108002 lui (15)
24c50004 addiu (9)
0060f809 (null) (0)
‘- jalr  $v1, $zero, $ra (shamt=0)
...
```

Fig. 3. Parte da saída gerada pelo desmontador em 30 de setembro de 2006

```
a.out:    file format elf32-tradbigmips
Disassembly of section .text:
80020000 <_ftext>:
80020000: 27bdffe0  addiu sp,sp,-32
80020004: afb10014  sw s1,20(sp)
80020008: 3c118002  lui s1,0x8002
8002000c: 922201b8  lbu v0,440(s1)
80020010: afbf0018  sw ra,24(sp)
80020014: 14400012  bnez v0,80020060
80020018: afb00010  sw s0,16(sp)
8002001c: 10000004  b 80020030
80020020: 3c108002  lui s0,0x8002
80020024: 24c50004  addiu a1,a2,4
80020028: 0060f809  jalr v1
...
```

Fig. 4. Parte da saída gerada pelo GNU `objdump`

com binários que modifiquem seu fluxo de controle durante a execução, como ocorre em programas que utilizam ponteiros para função, ou outras construções que dependam significativamente do ambiente de execução, como tratamento de exceções pelo sistema operacional.

Como exemplo, binários para diversas arquiteturas (Intel `x86`, por exemplo) utilizam a combinação de seções `.plt`[10] e `.got`[11] , ambas empregadas durante a ligação dinâmica de bibliotecas de que o binário necessita para executar. Esta ligação ocorre em tempo de execução, e o conteúdo de ambas as seções é modificado no transcorrer da execução [22].

Uma abordagem dinâmica, como proposta em diversos trabalhos relacionados ([9], [18], [19], [20], [21]), permitiria tratar um espectro mais abrangente de problemas, mas ainda assim não seria completa. Interações com o sistema operacional, por exemplo, precisariam ser emuladas, exigindo que a abordagem fique nos moldes de uma infra-estrutura de emulação completa, realizando inclusive a execução simbólica do binário sendo desmontado e bibliotecas de que ele necessite.

[10] `.plt` é a tabela para ligação de procedimentos ou *Procedure Linkage Table*, em Inglês.

[11] `.got` é a tabela de deslocamentos globais ou *Global Offset Table*, em Inglês.

VI. Conclusão e Trabalhos Futuros

Este relatório apresentou o andamento do trabalho proposto, tanto a parte investigativa quanto prática. Durante o período que passou, descobriu-se muito a respeito da área de engenharia reversa, que vem a ser a área do conhecimento que mais se aproxima tanto das técnicas aqui utilizadas quanto dos problemas aqui tratados. A preocupação em gerar código eficiente não é recente, e são justamente as técnicas de engenharia reversa que estão propiciando parte dos avanços mais significativos relacionados às novas técnicas de otimização. Isso fica ainda mais evidente quando se considera a vasta gama de trabalhos publicados na área.

Embora ferramentas para análise e modificação de binários em tempo de execução estejam em destaque, os problemas que podem ser tratados por ferramentas estáticas ainda são de grande relevância, como por exemplo a conversão entre conjunto de instruções similares mas com diferentes níveis de flexibilidade (o conjunto de instruções implementadas pelo processador MIPS R3000 é menor do que o R4000, embora muito similares).

Este trabalho adota técnicas atuais para lidar com a desmontagem, manipulação e remontagem de binários executáveis, além de propor uma extensão à técnica de desmontagem como forma de amenizar as dificuldades que surgem com o aparecimento de sombras no código.

Como trabalhos futuros, será necessário verificar a eficácia e validade desta extensão proposta, bem como investigar técnicas de otimização para que seja possível validar a infra-estrutura de manipulação de binários executáveis. Também será importante identificar de forma mais precisa o campo de aplicação da infra-estrutura proposta, apesar de já se saber que ela não será capaz de lidar com binários que alterem seu próprio código, ou que sejam muito dependentes do ambiente de execução ou de interação com o sistema operacional.

Além disso, espera-se que a infra-estrutura proposta permita a análise de interação entre as técnicas de otimização existentes que venham a ser implementadas como módulos para ela, e também o desenvolvimento e validação de novas técnicas de otimização.

Referências

[1] J. Backus, “The history of FORTRAN I, II, and III,” in *Proceedings of the 1st ACM SIGPLAN Conference on History of Programming Languages*, Los Angeles, CA, 1978, pp. 165–180.

[2] Z. Pan and R. Eigenmann, “Fast and effective orchestration of compiler optimizations for automatic performance tuning,” in *Proceedings of the International Symposium on Code Generation and Optimization (CGO’06)*, 2006, pp. 322–333.

[3] “GCC, the GNU compiler collection,” Nov. 2006. [Online]. Available: http://gcc.gnu.org/

[4] M. E. Benitez and J. W. Davidson, “The advantages of machine-dependent global optimization,” in *Proceedings of the Conference on Programming Languages and System Architectures*, 1994, pp. 105–124. [Online]. Available: http://citeseer.ist.psu.edu/benitez94advantages.html

[5] M. E. Benitez and J. Davidson, “Target-specific global code improvement: Principles and applications,” University of Virginia, Charlottesville, VA, EUA, Tech. Rep. CS-94-42, 1994. [Online]. Available: http://citeseer.ist.psu.edu/benitez94targetspecific.html

[6] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, 1994. [Online]. Available: http://doi.acm.org/10.1145/193209.193217

[7] S. W. K. Tjiang and J. L. Hennessy, "Sharlit–a tool for building optimizers," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*. New York, NY, USA: ACM Press, 1992, pp. 82–93. [Online]. Available: http://doi.acm.org/10.1145/143095.143120

[8] E. Rohou, F. Bodin, A. Seznec, G. L. Fol, F. Charot, and F. Raimbault, "SALTO: System for assembly-language transformation and optimization," IRISA, Tech. Rep. PI-1032, 1996. [Online]. Available: ftp://ftp.irisa.fr/techreports/1996/PI-1032.ps.gz

[9] E. Duesterwald, "Design and engineering of a dynamic binary optimizer," *Proceedings of the IEEE*, vol. 93, pp. 436–448, 2005. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1386661

[10] C. Cifuentes and V. Malhotra, "Binary translation: Static, dynamic, retargetable?" in *Proceedings of the International Conference on Software Maintenance (ICSM 1996)*, 1996, pp. 340–349. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=565037

[11] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*, 2002, pp. 45–54. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1173063

[12] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 255–270. [Online]. Available: http://www.usenix.org/events/sec04/tech/kruegel.html

[13] C. Cifuentes, "Interprocedural data flow decompilation," University of Queensland, Tech. Rep. FIT-TR-1994-04, 1994. [Online]. Available: http://portal.acm.org/citation.cfm?id=869854

[14] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Software–Practice and Experience*, vol. 25, pp. 811–829, 1995.

[15] C. Cifuentes and A. Fraboulet, "Intraprocedural static slicing of binary executables," in *Proceedings of the International Conference on Software Maintenance (ICSM 1997)*, 1997, pp. 188–195. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=624245

[16] Ákos Kiss, J. Jász, G. Lehotai, and T. Gyimóthy, "Interprocedural static slicing of binary executables," in *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, 2003, pp. 118–127. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1238038

[17] G. Stitt and F. Vahid, "New decompilation techniques for binary-level co-processor generation," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2005)*, 2005, pp. 547–554. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1560127

[18] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO 2003)*, 2003, pp. 265–275. [Online]. Available: http://dx.doi.org/10.1109/JPROC.2004.840302

[19] D. Ung and C. Cifuentes, "Machine-adaptable dynamic binary translation," in *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, 2000, pp. 41–51. [Online]. Available: http://portal.acm.org/citation.cfm?id=351397.351414

[20] S. Nanda, W. Li, L.-C. Lam, and T.-C. Chiueh, "BIRD: Binary interpretation using runtime disassembly," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*, 2006, pp. 359–370.

[21] D. Ung and C. Cifuentes, "Dynamic re-engineering of binary code with run-time feedbacks," in *Proceedings of the 7th Working Conference on Reverse Engineering*, 2000, pp. 2–10. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=891447

[22] *Executable and Linking Format (ELF) Specification Version 1.2*, May 1995. [Online]. Available: http://x86.ddj.com/ftp/manuals/tools/elf.pdf

[23] M. L. Scott, *Programming Language Pragmatics*. San Diego, CA, EUA: Academic Press, 2000, ch. 9, pp. 492–496.

[24] *MIPS32® Architecture For Programmers*. Mountain View, CA, EUA: MIPS Technologies, Inc., July 2005, vol. 2, revision 2.50. [Online]. Available: http://www.mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/doclibrary