Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anže Žitnik

Orodje za delo z objektnimi datotekami

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Tomaž Dobravec

Ljubljana 2012

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00018/2012
Datum: 05.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ANŽE ŽITNIK**

Naslov: **ORODJE ZA DELO Z OBJEKTNIMI DATOTEKAMI**
OBJECT FORMAT MANAGER

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Preglejte obstoječe formate objektnih datotek in jih med seboj primerjajte. Posebno pozornost posvetite formatu ELF, natančno preglejte njegovo strukturo in se spoznajte s posameznimi sestavnimi deli (glave, sekcije, segmenti, polja, ...). Izdelajte javanski paket, ki bo omogočal ustvarjanje, branje, spreminjanje in zapisovanje datotek v formatu ELF. Ustvarite metode za preprost dostop do omenjene funkcionalnosti. S pomočjo razvitega paketa izdelajte program, ki bo uporabniku omogočal pregledovanje datotek ELF. Preučite možnost spreminjanja statičnih nizov (literalov) v datotekah ELF in dopolnite program tako, da bo uporabnik to operacijo izvajal na čimbolj enostaven način.

Mentor:
doc. dr. Tomaž Dobravec

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Anže Žitnik, z vpisno številko **63090143**, sem avtor diplomskega dela z naslovom:

Orodje za delo z objektnimi datotekami

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 31. avgusta 2012 Podpis avtorja:

Za vodenje in pomoč se zahvaljujem mentorju dr. Tomažu Dobravcu.

Kazalo

Povzetek

Glavni cilj te diplomske naloge je bil podrobno spoznavanje formata objektnih datotek ELF in razvoj javanskega paketa za preslikavo vsebine datotek ELF v javanske objekte. Pregledali smo različne formate zapisov objektnih datotek in pri tem podrobneje obravnavali format ELF. Proučili smo možnosti pretvorbe izvršljivih datotek iz enega formata v drugega. Izdelali smo javanski paket, ki omogoča branje, pisanje, spreminjanje in ustvarjanje novih datotek ELF brez začetne osnove. Razvili smo metodo za urejanje znakovnih nizov v izvršljivih objektnih datotekah. Paket smo uporabili za izdelavo aplikacije, ki omogoča pregled vsebine datotek ELF in, s pomočjo omenjene metode, tudi polavtomatsko prevajanje nizov v izvedljivih programih.

Ključne besede

objektne datoteke, format ELF, urejanje nizov v izvršljivih datotekah

Abstract

The main goal of this thesis was a detailed study of the ELF object file format and development of a Java package for mapping the contents of ELF files to Java objects. We reviewed various object file formats and examined the ELF format in greater detail. We studied the possibilities of converting executable files from one file format to another. We made a Java package that enables reading, writing, modifying and creating new ELF files without an existing base. We developed a method for editing character strings in executable object files. We used the package to develop an application that provides an overview of ELF files' contents and, with support of the before mentioned method, semi-automatic translation of strings in executable programs.

Keywords

object files, ELF format, editing strings in executable files

Poglavje 1

Uvod

Z objektnimi datotekami se v računalništvu srečujemo na vsakem koraku, pogosto pa se ne zavedamo, kaj vse se v njih skriva. Razumevanje zgradbe objektnih datotek pripomore k razumevanju delovanja povezovalnikov, nalagalnikov in druge sistemske programske opreme, poleg tega pa so nekatere metode za zapis podatkov v objektnih datotekah uporabne pri reševanju drugih računalniških problemov.

Cilj te diplomske naloge je bil razviti orodje za delo z objektnimi datotekami formata ELF. Razvili smo javanski paket, ki vsebino datotek ELF predstavi z javanskimi objekti in nudi metode za branje, pisanje in spreminjanje posameznih delov ali celotnih datotek ELF. Na osnovi tega paketa smo ustvarili program z grafičnim uporabniškim vmesnikom, ki na enostaven in pregleden način prikazuje vsebino ELF datotek, vključuje pa tudi funkcijo za spreminjanje znakovnih nizov v izvršljivih datotekah. To funkcijo lahko uporabimo za prevajanje uporabniških vmesnikov programov, predstavlja pa posebej zanimiv del te diplomske naloge, saj smo ob njenem razvoju naleteli na nemalo težav. Raziskali smo tudi možnosti prevedbe izvršljivih datotek iz enega formata v drugega. Namen take prevedbe bi bil izvajanje programa na operacijskem sistemu, drugačnem od tistega, za katerega je bil program v času prevajanja namenjen.

V tem delu najprej predstavimo različne namene objektnih datotek in nji-

hovo zgradbo. Nato pregledamo lastnosti nekaj popularnih formatov objektnih datotek, pri tem pa se podrobneje posvetimo formatu ELF. Opišemo tudi možnosti in smiselnost pretvorbe izvršljivih datotek med formati. V poglavju 3 predstavimo zgradbo ter opišemo zanimivejše funkcionalnosti izdelanega javanskega orodja za delo z datotekami ELF. V poglavju 4 pa si pogledamo našo implementacijo programa za pregled objektnih datotek ELF ter predstavimo delovanje ter uspešnost metode za urejanje nizov v izvršljivih programih.

Poglavje 2

Objektne datoteke

Objektna datoteka (tudi objektni modul) je datoteka, ki vsebuje strojno kodo za izvajanje na nekem računalniku. Največkrat je objektna datoteka izhod zbirnika (*assembler*) ali prevajalnika (*compiler*). Uporabljamo jih za shranjevanje prevedenih programov, poleg same strojne kode pa lahko vsebujejo tudi reference za povezovanje z drugimi objektnimi datotekami, informacije za prenaslavljanje (*relocation*), razne komentarje, informacije za razhroščevanje (*debugging*) in drugo.

Objektne datoteke lahko glede na namen uporabe razdelimo na povezljive (*linkable*), izvršljive (*executable*) in naložljive (*loadable*), posamezna datoteka pa je lahko uporabljena tudi kot poljubna kombinacija teh treh. **Povezljiva** datoteka je uporabna le za povezovanje z drugimi objektnimi datotekami, **izvršljiva** datoteka je lahko naložena v pomnilnik in zagnana kot program, **naložljiva** datoteka pa je lahko naložena v pomnilnik poleg programa ter uporabljena kot knjižnica.

Za razumevanje uporabe in formatov objektnih datotek je koristno, da razumemo tudi osnove povezovalnikov in nalagalnikov. Povezovalnik (*linker*) in nalagalnik (*loader*) sta programa, katerima so objektne datoteke primarno namenjene in sta del vsakega sodobnega operacijskega sistema. V podrobnosti se v tem delu ne bomo spuščali, več pa si lahko preberete v [2].

Glavna naloga povezovalnikov je prireditev konkretnih vrednosti bolj ab-

straktnim imenom, kar omogoča programerjem pisati kodo z uporabo abstraktnih imen. Tako lahko na primer programer kliče neko funkcijo (ki je definirana v drugi objektni datoteki) z njenim simboličnim imenom, povezovalnik pa to ime prevede v dejanski naslov te funkcije, uporaben procesorju. Poleg tega pa povezovalniki združujejo povezljive objektne datoteke v izvršljive, naložljive ali nove povezljive datoteke. Posebna vrsta povezovalnika je dinamični povezovalnik (*dynamic linker*), ki med izvajanjem programa naloži v pomnilnik in poveže knjižnice, ki jih izvajan program potrebuje.

Nalagalnik skrbi za nalaganje v pomnilnik in zagon (skok na začetni naslov) programa. Če je potrebno, nalagalnik, v skladu z zapisi za prenaslavljanje, tudi popravi potrebne naslove v programu glede na naslov nalaganja.

2.1 Sestavni deli objektnih datotek

Objektne datoteke lahko sestavljajo naslednji podatki:

Glava Vsebuje metapodatke o datoteki, kot so: vrsta, velikost in odmik posameznih delov, začetni naslov programa, čas nastanka ...

Strojna koda Vsebuje binarne strojne ukaze in podatke, ki jih generira prevajalnik ali zbirnik. Je obvezen del smiselne objektne datoteke. Sekcija z ukazi se navadno imenuje `text`, sekcija z inicializiranimi podatki pa `data`.

Vnosi za prenaslavljanje Določajo mesta v strojni kodi, ki morajo biti popravljena ob spreminjanju naslova, kamor je program naložen. V prenaslovljivi datoteki ti vnosi služijo tudi za označevanje referenc na nedefinirane simbole. Tako lahko povezovalnik ob vključitvi simbola na ta mesta vstavi njegov dejanski naslov.

Seznam simbolov Označuje simbole, ki so definirani v tej datoteki in simbole iz drugih datotek, ki jih program v tej datoteki potrebuje.

Informacije za razhroščevanje Vsebujejo podatke, ki sicer niso pomembni za normalno izvajanje programa, temveč za razhroščevanje. To so podatki o izvorni kodi, na primer: številke vrstic, simbolična imena, podatki o uporabljenih podatkovnih strukturah ...

Sekcija `bss` Posebna sekcija, ki pove nalagalniku, na katerem mestu v pomnilniku in koliko prostora naj rezervira za neinicializirane statične spremenljivke programa. Pred zagonom programa se tja zapišejo ničle. Pri nekaterih enostavnejših formatih so ničle, ki predstavljajo sekcijo `bss`, fizično zapisane v datoteki in se naložijo skupaj s kodo.

Objektna datoteka lahko vsebuje tudi druge tipe podatkov, ki pa niso tako pomembni. Posamezni deli datoteke so v večini formatov razdeljeni po sekcijah določenih tipov. Programska koda je vsebovana v segmentih. Segmenti so, kjer se uporablja pomnilniško ostranjevanje (*paging*), ponavadi poravnani na naslov strani zaradi enostavnejšega pomnilniškega preslikovanja (*memory mapping*).

2.2 Formati objektnih datotek

Zasnova formata objektnih datotek se razlikuje glede na namen datotek, za katere je zasnovan. Kot smo že omenili, poznamo povezljive, izvršljive in naložljive objektne datoteke. Nekateri formati podpirajo le enega ali dva od teh namenov, drugi pa vse tri. Razlog za razliko med formati za različne namene je, da se v večini primerov logično grupiranje delov kode za povezovanje razlikuje od tistega za nalaganje programa. Navadno povezovalnik bere objektno datoteko po delih, nalagalnik pa naloži celotno datoteko naenkrat (ali po večjih kosih) v pomnilnik.

Pogledali si bomo osnovne značilnosti nekaj konkretnih formatov objektnih datotek, v razdelku 2.3 pa bo podrobneje predstavljen objektni format ELF. Poleg spodaj opisanih pa poznamo še objektne formate COFF, Mach-O, PEF, OMF in druge. Nekaj jih je opisanih v [2].

2.2.1 Objektni format COM

Objektni format COM (*Command file*) je najenostavnejši format objektnih datotek. Primeren je le za izvršljive datoteke. Vsebuje namreč le izvedljivo strojno kodo, brez kakršnihkoli metapodatkov. Pri zagonu programa operacijski sistem enostavno naloži vsebino datoteke v pomnilnik na naslov $100_{[16]}$ (na naslovih od 0 do $FF_{[16]}$ je tako imenovani PSP (*Program Segment Prefix*) z argumenti programa in drugimi parametri), nastavi potrebne registre ter skoči na začetek programa.

Ker se program vedno naloži na isti naslov v pomnilniku, datoteka COM za programe krajše od dolžine segmenta ne potrebuje zapisov za prenaslavljanje. Če se program ne prilega v en pomnilniški segment[1], mora programer sam poskrbeti za prenaslavljanje. Prav take nevšečnosti pa odpravljajo današnji povezovalniki in nalagalniki.

COM datoteke so se uporabljale v operacijskem sistemu Microsoft DOS, na procesorjih arhitekture Intel x86. Na sodobnih sistemih Microsoft Windows se lahko programi v COM datotekah izvajajo le v emuliranem DOS okolju. Objektne datoteke tega formata imajo imena s končnico `.COM`. Najbolj znana datoteka formata COM je lupina sistema MS-DOS, COMMAND.COM, ki pa je bila v novejših verzijah operacijskega sistema (čeprav je ohranila ime) zapisana v formatu MZ.

2.2.2 Objektni format a.out

Večina sodobnih operacijskih sistemov deluje tako, da za vsak na novo zagnan program ustvari lasten naslovni prostor. V tem primeru so lahko programi zasnovani za nalaganje na fiksen naslov in ne potrebujejo prenaslavljanja ob nalaganju. Takim situacijam je namenjen format a.out (*Assembler Output*). Primeren je tako za izvršljive kot tudi za povezljive in naložljive datoteke.

Obstaja pet različnih variant formata a.out, ki so se uporabljale za ra-

[1] Ker je velikost datoteke COM omejena na velikost segmenta (64kB), mora za nalaganje večjih programov program vsebovati lasten nalagalnik.

zlične vrste objektnih datotek ter v različnih obdobjih (novejše različice so zamenjale starejše). Vsaka varianta ima svojo čarobno številko (*magic number*). Za zanimivost povejmo, da je čarobna številka najstarejše variante, $407_{[8]}$, na računalniku PDP-11[2] pomenila ukaz za skok 7 besed naprej (na začetek sekcije s strojno kodo), kar je dovoljevalo nalagalnikom začeti izvajanje programa na odmiku 0 od začetka datoteke.

Vsaka datoteka formata a.out ima na začetku 8 besed dolgo glavo, ki vsebuje čarobno številko, velikost posameznih sekcij ter vstopno točko programa (naslov prvega ukaza). Vrsta sekcij in njihov vrstni red v datoteki je vnaprej določen: sekcija s programsko kodo (`text`), inicializirani podatki (`data`), tabela simbolov ter tabeli z zapisi za prenaslavljanje, posebej za sekciji `text` in `data`. Sekcija `bss` ne zaseda prostora v datoteki, njena velikost v pomnilniku pa je prav tako zapisana v glavi.

Datoteke formata a.out so se uporabljale v starejših verzijah operacijskih sistemov družine UNIX na različnih procesorskih arhitekturah. Zaradi omejitev pri številu in vrsti sekcij ter zaradi pomanjkanja podpore dinamičnim nalagalnikom je bil v sistemu *Unix System V* format a.out zamenjan s formatom COFF (*Common Object File Format*), kasneje pa s formatom ELF. Format a.out je bil uporabljan tudi v operacijskem sistemu Linux do verzije jedra 1.2 (leta 1995), ko je bil nadomeščen s formatom ELF. Danes je od formata a.out ostalo privzeto ime izvršljivih datotek nekaterih prevajalnikov (gcc), čeprav so te v drugem formatu.

2.2.3 Objektni format DOS MZ

Objektni format MZ[3] (z drugim imenom format DOS EXE) je v operacijskem sistemu Microsoft DOS nasledil objektni format COM. Prednost tega formata je predvsem v podpori zapisom za prenaslavljanje.

Kljub temu, da vsakemu programu ustvarijo lasten naslovni prostor, neka-

[2]Serija računalnikov podjetja DEC (*Digital Equipment Corporation*).

[3]Ime in čarobna številka formata (ASCII koda za znaka "MZ") izvirata iz začetnic enega od snovalcev sistema Microsoft DOS, Marka Zbikowskega.

teri operacijski sistemi (tudi 32-bitne različice sistemov Windows), ne naložijo programov vedno na isti naslov. V tem primeru morajo izvršljive datoteke vsebovati zapise za prenaslavljanje. Ti označujejo mesta, kjer je treba naslove v programu spremeniti glede na naslov nalaganja.

Kot tudi pri objektnem formatu COM, DOS naloži program v strnjen blok prostega pomnilnika. Program naslavlja pomnilnik s številko segmenta in z naslovom znotraj segmenta. Če program presega dolžino enega segmenta, mora za naslavljanje eksplicitno podajati številke zahtevanih segmentov. Ker je program premaknjen kot celota, se odmiki znotraj segmenta ne spreminjajo. Ob nalaganju programa mora zato nalagalnik spremeniti le številke segmentov v kodi programa. V datoteki so številke segmentov shranjene kot da bo program naložen v pomnilnik na lokacijo 0.

Ta format je en najenostavnejših objektnih formatov z zapisi za prenaslavljanje. Poleg programske kode vsebuje le še glavo z osnovnimi metapodatki ter tabelo z zapisi za prenaslavljanje. Nalaganje programov v objektnih datotekah MZ je le malo bolj zapleteno od nalaganja programa v formatu COM, razlikuje se predvsem v tem, da je treba pregledati zapise za prenaslavljanje in ustrezno popraviti kodo programa.

Objektni format MZ se je uporabljal za izvršljive datoteke na operacijskih sistemih Microsoft DOS in Windows do verzije Windows NT 3.1, programe v tem formatu pa lahko na sodobnih sistemih zaženemo v raznih DOS emulatorjih. Imena datotek objektnega formata MZ imajo končnico `.exe`.

2.2.4 Objektni format PE

Microsoftov objektni format PE (*Portable Executable*) izvira iz formata COFF, ki so ga uporabljali sistemi Unix za formatom a.out in pred formatom ELF. Z operacijskim sistemom Windows NT 3.1 je nasledil objektni format MZ. V nasprotju s prejšnjima opisanima Microsoftovima formatoma (COM in MZ) so objektne datoteke foramta PE lahko izvršljive ali naložljive. Izvršljive imajo imena s končnico `.exe`, naložljive pa `.dll` in jim pravimo tudi dinamične knjižnice DLL (*Dynamic Link Library*). PE datoteke so namenjene

uporabi v okolju s pomnilniškim ostranjevanjem, tako da se lahko strani enostavno preslikajo iz datoteke v pomnilnik, podobno kot bomo videli pri objektnem formatu ELF.

Datoteka formata PE ima na začetku vsebino majhne datoteke objektnega formata MZ.[4] Tam je v formatu MZ shranjen kratek program, ki izpiše obvestilo[5], da program v tej datoteki ne more biti zagnan na tem operacijskem sistemu. Tako je zagotovljeno, da v operacijskem sistemu, ki uporablja objektni format MZ, uporabnik ob zagonu datoteke PE dobi (vsaj malo) uporabno obvestilo namesto tistega o neznani datoteki.

Programu v formatu MZ sledi podpis, glava formata COFF in glava formata PE, ki vsebuje različne metapodatke. Temu sledi tabela glav sekcij s podatki o sekcijah, kot so ime, velikost v datoteki, velikost v pomnilniku, odmik v datoteki, naslov v pomnilniku in zastavice s parametri. Velikost sekcije v pomnilniku se lahko razlikuje od velikosti v datoteki. Velikost pomnilniške strani je ponavadi večja od velikosti bloka na disku in sekcija, ki se konča na sredini strani, ne potrebuje blokov na disku do konca strani. To prihrani manjšo količino prostora na disku. Naslovi v glavah datotek formata PE (naslov sekcije v pomnilniku ipd.) so navedeni kot odmik od navideznega naslova v pomnilniku, kamor se program naloži.

Posebne sekcije formata PE

Datoteke formata PE vsebujejo nekaj specifičnih tipov sekcij, ki so zanimive zaradi oblike shranjevanja podatkov:

Izvozni simboli Seznam simbolov, definiranih v tej datoteki in vidnih drugim datotekam. Izvršljive datoteke navadno nimajo izvoznih simbolov, datoteke knjižnic DLL pa izvažajo simbole za funkcije in podatke, ki jih nudijo. Na simbol se lahko sklicujemo z imenom ali pa s celoštevilskim indeksom. Tabela z imeni simbolov je urejena po abecedi, da omogoča

[4] Začetnice Marka Zbikowskega tako ostajajo kot prva dva znaka tudi v današnjih izvršljivih datotekah operacijskih sistemov Windows.

[5] Ponavadi "*This program cannot be run in DOS mode.*"

iskanje z bisekcijo. Poleg identifikacije simbola je njegov navidezni naslov.

Uvozni simboli Seznam simbolov, ki morajo biti ob nalaganju razrešeni iz knjižnic DLL. Posamezen simbol je lahko identificiran z imenom ali pa s številsko vrednostjo. Če je podano ime simbola, je podano tudi število, ki predvideva mesto v tabeli izvoznih simbolov vhodne knjižnice. S tem se lahko iskanje pohitri. Poleg identifikacije simbola je navidezni naslov, na katerega naj se naloži njegova vrednost.

Viri Razni podatki v obliki, specifični za posamezen tip vira. Sem spadajo grafična ikona programa, posebni slogi pisav, bitne slike, ki so del grafičnega uporabniškega vmesnika in podobno. Posamezni viri so predstavljeni kot listi v drevesu, kjer vozlišča na prvem nivoju vsebujejo tip vira, na drugem ime vira, na tretjem nivoju pa oznako jezika vira. Oznaka jezika predstavlja naravni jezik (npr. slovenščina) in programu omogoča prilagajanje glede na želen jezik uporabnika.

Vnosi za prenaslavljanje Povezovalnik ustvari datoteko PE za določen ciljni naslov v pomnilniku, kamor bo datoteka preslikana. Lahko pa se zgodi, da program ne more biti naložen na predvideno lokacijo. To se velikokrat zgodi pri dinamičnih knjižnicah DLL in, v nekaterih primerih, tudi pri izvršljivih datotekah. V tem primeru mora datoteka PE vsebovati zapise za prenaslavljanje. Sekcija s temi zapisi vsebuje bloke zapisov, kjer vsak blok nosi zapise za eno 4096 bajtov veliko stran programa. Blok vsebuje predviden naslov strani, število zapisov in seznam 16 bitnih zapisov za prenaslavljanje. Spodnjih 12 bitov zapisa predstavlja odmik znotraj strani, kjer je treba narediti prenaslavljanje, zgornji 4 biti pa povedo, kako dolg je naslov, ki ga je treba spremeniti (spremeni 32 bitov, zgornjih 16 bitov ali spodnjih 16 bitov). Taka podatkovna struktura vnosov za prenaslavljanje lahko prihrani nekaj prostora, saj je en vnos predstavljen le z dvema bajtoma (pri objektnem formatu ELF je en vnos velik 8 oz. 12 bajtov).

Več o objektnem formatu PE si lahko preberete v [4] in [5].

2.3 Objektni format ELF

Objektni format ELF (*Executable and Linking Format*) je naslednik formata COFF na večini operacijskih sistemov družine Unix. Objektni format COFF je bil zamenjan, ker ni deloval dobro na sistemih s časovnim dodeljevanjem (*time-sharing*) in je imel slabo podporo dinamičnemu povezovanju. Na operacijskem sistemu Linux se format COFF ni uporabljal in je ELF direktno zamenjal format a.out. Še vedno pa Linuxov prevajalnik gcc privzeto poimenuje izhodne datoteke z imenom a.out, čeprav so v formatu ELF.

Kot bomo spoznali v tem poglavju, je format ELF v primerjavi z ostalimi opisanimi formati relativno kompliciran, a zelo vsestranski in prilagodljiv. Zato se uporablja na mnogih operacijskih sistemih, kot so: Linux, Solaris, FreeBSD, OpenBSD, Syllable, HP-UX, OpenVMA, AmigaOS 4, MorphOS, Symbian OS 9, Bada... in na najrazličnejših procesorskih arhitekturah. Uporabljajo ga tudi igralne konzole (med drugimi Sony PlayStation, Nintendo GameCube, Nintendo Wii) ter nekateri mobilni telefoni.

Obstajata dve različici objektnega formata ELF, 32-bitna in 64-bitna. Razlike med njima so minimalne, 64-bitna različica le razširja različna polja v formatu. Originalno specifikacijo objektnega formata ELF najdete v [10] in [11], najnovejšo verzijo pa v [8].

Datoteke formata ELF se lahko uporabljajo kot povezljive, izvršljive ali naložljive. V specifikaciji ELF jim pravimo (po vrsti) prenaslovljive (*relocatable*), izvršljive (*executable*) in deljene (*shared object*). Prenaslovljive datoteke so uporabne za vhod povezovalniku, izvršljive imajo razrešene vse simbole (razen morda nekaterih, ki pripadajo dinamičnim knjižnicam) in se lahko poženejo, deljene datoteke pa vsebujejo knjižnice s simboli za povezovanje in izvedljivo kodo. Zato, da so datoteke ELF za vse namene v enakem formatu enostavno in učinkovito razumljive vsem vpletenim programom, imajo značilno strukturo, sestavljeno iz sekcij in segmentov.

2.3.1 Struktura datoteke formata ELF

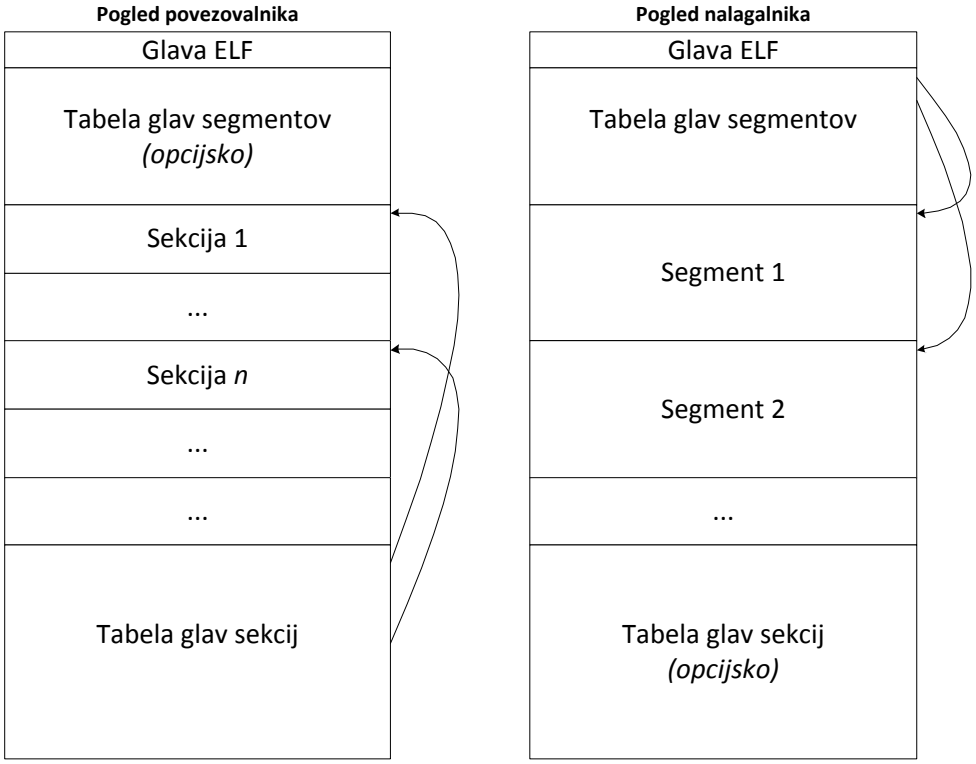
Prevajalniki, zbirniki in povezovalniki obravnavajo vsebino datotek ELF kot množico sekcij, nalagalniki pa kot množico segmentov. Sekcije so opisane s tabelo glav sekcij (*section header table*), segmenti pa s tabelo glav segmentov (*program header table*). Sekcije so namenjene nadaljnji obdelavi s povezovalnikom, segmenti pa preslikavi v pomnilnik. En segment ponavadi obsega več sekcij. Na primer; segment označen kot naložljiv in samo za branje vsebuje sekcije za izvršljivo kodo, podatke samo za branje in simbole za dinamično povezovanje. Vse objektne datoteke ELF lahko imajo sekcije in segmente, ni pa to vedno potrebno. Prenaslovljive datoteke ne potrebujejo segmentov, ker se z njimi nalagalniki ne ukvarjajo. Izvršljive datoteke praktično tudi ne potrebujejo sekcij, vendar je v specifikaciji formata ELF določeno, da so vsi podatki v datoteki (razen glav) vsebovani v sekcijah. Zato naj bi sekcije imele vse (smiselne) datoteke ELF. Struktura objektne datoteke ELF ter pogled povezovalnika in pogled nalagalnika so prikazani na sliki 2.1.

Glava ELF

Datoteka ELF se začne z glavo (*ELF header*), ki vsebuje splošne podatke o datoteki:

e_ident Identifikacijski podatki. Polje, ki vsebuje podatke, pomembne za pravilno branje preostanka datoteke:

- **EI_MAG** Čarobna številka. Prvi štirje bajti datoteke z vrednostmi $7F_{16}$, 'E', 'L', 'F', označujejo datoteko kot objektno datoteko formata ELF.
- **EI_CLASS** Razred. Pove, ali je objektna datoteka 32-bitnega ali 64-bitnega razreda. 64-bitni razred datotek se uporablja na 64-bitnih arhitekturah in ima zato nekatera polja daljša.
- **EI_DATA** Kodiranje podatkov. Določa način za shranjevanje sestavljenih pomnilniških operandov - pravilo debelega ali tankega konca.



Slika 2.1: Struktura objektne datoteke formata ELF z dveh zornih kotov. Povzeto po [10].

EI_VERSION Bajt, ki označuje verzijo formata identifikacijskih podatkov. Format ELF je zasnovan tako, da se lahko v prihodnosti razširi in podpre nove funkcionalnosti. Kljub temu, da specifikacija formata sega v leto 1995 in je bila do danes (2012) že deležna nekaj manjših sprememb, pa se še vedno uporablja enaka številka verzije, 1.

EI_OSABI Identifikacija operacijskega sistema in programskega vmesnika (ABI). Označuje, da so v datoteki uporabljene razširitve formata ELF, specifične za določen operacijski sistem ali programski vmesnik. Nekatera polja v datoteki imajo vrednosti, ki se interpretirajo glede na vrednost tega bajta.

EI_ABIVERSION Verzija programskega vmesnika, opisanega s prejšnjim bajtom.

EI_PAD Neuporabljen prostor. Trenutna verzija formata ELF na tem mestu vsebuje 7 neuporabljenih bajtov, ki so namenjeni prihodnjim razširitvam formata.

e_type Tip objektne datoteke. Najpogosteje je to prenaslovljiva, izvršljiva ali deljena datoteka, lahko pa zavzame tudi drugo vrednost, specifično operacijskemu sistemu.

e_machine Arhitektura računalnika, kateremu je namenjena programska koda v objektni datoteki.

e_version Število, ki označuje verzijo formata ELF, uporabljeno v datoteki. Podobno kot pri verziji formata identifikacijskih podatkov, se tudi tukaj uporablja število 1.

e_entry Vstopna točka programa. Predstavlja navidezni naslov, na katerem naj se začne izvajanje vsebovanega programa.

e_phoff Odmik (v bajtih) v datoteki, kjer se nahaja tabela glav segmentov.

e_shoff Odmik v datoteki, kjer se nahaja tabela glav sekcij.

e_flags Polje, ki vsebuje zastavice, povezane z datoteko. Pomen zastavic je specifičen tipu procesorja.

e_ehsize Velikost glave ELF. Ta je sicer vedno 52 bajtov za datoteke 32-bitnega razreda in 64 bajtov za datoteke 64-bitnega razreda. To polje pa je v glavi vseeno vsebovano zaradi možnih kasnejših sprememb standarda. Podobno velja za velikost glave segmenta in velikost glave sekcije.

e_phentsize Velikost glave segmenta.

e_phnum Število glav segmentov.

e_shentsize Velikost glave sekcije.

e_shnum Število glav sekcij. Če je število glav sekcij v datoteki večje ali enako $FF00_{16}$, je na tem mestu zapisano število 0, dejansko število sekcij pa je zapisano v polju sh_size glave sekcije z indeksom 0.

e_shstrndx Število, ki označuje indeks sekcije, ki vsebuje nize z imeni sekcij. Če je indeks večji ali enak $FF00_{16}$, je na tem mestu zapisano število $FFFF_{16}$, dejanski indeks tabele nizov pa je zapisan v polju sh_link glave sekcije z indeksom 0.

2.3.2 Sekcije

Povezovalniki vidijo objektno datoteko ELF kot skupek sekcij. Vsaka sekcija vsebuje določen tip podatkov, na primer strojno kodo, podatke samo za branje, podatke za branje in pisanje ali vnose za prenaslavljanje. Tip sekcije in ostali metapodatki so zapisani v glavi sekcije (*section header*), ki se nahaja v tabeli glav sekcij. Podatki sekcije obsegajo en neprekinjen blok v objektni datoteki, ki pa ima lahko dolžino nič (na primer sekcija `bss`). Podatki dveh sekcij se ne smejo prekrivati, objektna datoteka pa lahko ima tudi neizkoriščen prostor. To je prostor, ki ni del nobene sekcije in vsebuje nedefinirane vrednosti.

Glave sekcij

Tabela glav sekcij se navadno nahaja na koncu datoteke ELF, pri tem pa sicer ni nobene omejitve. Lahko se nahaja kjerkoli v datoteki (odmik je zapisan v glavi ELF). Sestavljajo jo glave sekcij, njihovo število je zapisano v glavi ELF. To so vnosi, ki vsebujejo podatke o vsaki sekciji v datoteki. Ti, med drugim, omogočajo, da v datoteki najdemo vsako od sekcij. Čeprav pa ima vsaka prisotna sekcija svojo glavo, pa ni nujno, da ima vsaka glava svojo sekcijo. Nekatere sekcije namreč v datoteki ne potrebujejo nobenih podatkov razen glave, poseben primer pa tudi je glava sekcije z indeksom 0, ki je rezervirana in, v nekaterih primerih, vsebuje splošne podatke o datoteki.

Posamezna glava sekcije vsebuje naslednja polja:

sh_name Ime sekcije. Vrednost tega polja je odmik v sekciji, ki vsebuje nize z imeni sekcij, kjer se začne niz z imenom te sekcije. Sekcijo z imeni sekcij najdemo preko njenega indeksa, ki je zapisan v glavi ELF.

sh_type Tip sekcije. Glede na vrednost tega polja lahko interpretiramo vsebino sekcije. Nekateri tipi sekcij:

- **SHT_NULL** V resnici ni tip sekcije, ampak oznaka, da glava sekcije, ki jo vsebuje, nima pripadajoče sekcije.
- **SHT_PROGBITS** Tip sekcije za podatke programa. Pomen in format podatkov je določen le s programom. Sekcije tega tipa se uporabljajo za programsko kodo in podatke.
- **SHT_SYMTAB in SHT_DYNSYM** Tipa sekcij, ki označujeta simbolno tabelo. Objektna datoteka ima lahko samo eno sekcijo vsakega od teh dveh tipov. Simbolne tabele vsebujejo podatke, potrebne za lociranje simbolov, ki jih definira ali uvaža ta objektna datoteka.
- **SHT_STRTAB** Tip sekcije, ki vsebuje tabelo znakovnih nizov. Med drugim se uporablja za sekcijo, ki vsebuje imena sekcij.
- **SHT_REL in SHT_RELA** Tipa sekcij z vnosi za prenaslavljanje.

SHT_DYNAMIC Tip sekcije, ki vsebuje podatke, potrebne za dinamično povezovanje.

SHT_NOTE Tip sekcije, ki vsebuje posebne podatke o objektni datoteki. To je, na primer, ime in verzija prevajalnika, ki je ustvaril datoteko. Ti podatki so lahko uporabni za preverjanje kompatibilnosti med programi, največkrat pa ne vplivajo na uporabnost datoteke in vsebujejo le podpis prevajalnika oziroma povezovalnika.

SHT_NOBITS Tip sekcije, ki ne zaseda prostora v datoteki, sicer pa je podoben tipu SHT_PROGBITS. Uporablja se za sekcijo `bss`.

sh_flags To polje vsebuje zastavice, ki opisujejo atribute pripadajoče sekcije. Med drugim so to:

SHF_WRITE Podatki v tej sekciji imajo omogočeno pisanje med izvajanjem programa.

SHF_ALLOC Ta sekcija zaseda pomnilnik med izvajanjem programa.

SHF_EXECINSTR Ta sekcija vsebuje izvršljive strojne ukaze.

SHF_STRINGS Podatki te sekcije sestojijo iz nizov znakov.

sh_addr Naslov v pomnilniku, kamor bo naložena ta sekcija. Če sekcija ne bo del pomnilniške slike, to polje vsebuje vrednost 0.

sh_offset Odmik od začetka datoteke do začetka podatkov sekcije.

sh_size Velikost te sekcije v bajtih. Če je sekcija tipa SHT_NOBITS (ne glede na vrednost tega polja) ne zaseda prostora v datoteki.

sh_link Indeks sekcije, ki je povezana s to sekcijo. Interpretacija povezave je odvisna od tipa sekcije.

sh_info To polje vsebuje dodatno informacijo, katere interpretacija je odvisna od tipa sekcije.

sh_addralign Nekatere sekcije imajo omejitve glede poravnanosti. Če, na primer, sekcija vsebuje operande, velike 8 bajtov, mora sistem zagotoviti poravnanost na 8 bajtov za celotno sekcijo. Veljati mora:

$$\mathsf{sh\_addr} \equiv 0 \pmod{\mathsf{sh\_addralign}}$$

sh_entsize Nekatere sekcije vsebujejo vnose stalne velikosti, kot, na primer, simbolna tabela. Za take sekcije to polje vsebuje velikost posameznega vnosa. Sicer je vrednost polja 0.

Posebne sekcije

Iz imena lahko za posamezno sekcijo še natančneje izvemo, katere podatke vsebuje in kakšno vlogo ima. Nekatera imena sekcij so rezervirana (vsa se začnejo s piko) in jih za enak namen uporabljajo vsi programi, ki gradijo datoteke ELF. Sicer pa za delovanje objektnih datotek ELF imena sekcij niso pomembna. Posebno sekcijo prepoznamo po rezerviranem imenu in ujemajočem tipu ter zastavicah. Nekatere posebne sekcije so:

.bss Kot smo že omenili, sekcija `bss` predstavlja prostor za neinicializirane statične spremenljivke, ki ga nalagalnik napolni z ničlami. Kot nakazuje tip sekcije, SHT_NOBITS, ta sekcija ne zaseda prostora v datoteki. Aktivni sta zastavici SHF_ALLOC in SHF_WRITE.

.data Ta sekcija vsebuje inicializirane podatke programa. Tip sekcije je SHT_PROGBITS, aktivni ima zastavici SHF_ALLOC in SHF_WRITE.

.dynsym Ta sekcija vsebuje tabelo s simboli za dinamično povezovanje. Tip sekcije je SHT_DYNSYM, aktivno ima zastavico SHF_ALLOC.

.fini Ta sekcija vsebuje strojne ukaze, ki se izvedejo po končanju programa. Tip sekcije je SHT_PROGBITS, aktivni ima zastavici SHF_ALLOC in SHF_EXECINSTR.

.init Ta sekcija vsebuje strojne ukaze, ki se izvedejo pred začetkom izvajanja glavnega programa. Tip sekcije je SHT_PROGBITS, postavljeni ima zastavici SHF_ALLOC in SHF_EXECINSTR.

.interp Ta sekcija vsebuje znakovni niz, ki predstavlja pot do programa, ki se bo pognal kot interpreter. Če je ta sekcija prisotna, se, namesto direktnega zagona programa, zažene podani interpreter, kateremu je kot argument podana datoteka ELF. To se uporablja predvsem za zagon dinamičnega povezovalnika. Tip sekcije je SHT_PROGBITS, vsebovana pa je v segmentu tipa PT_INTERP. V nekaterih primerih ima postavljeno zastavico SHF_ALLOC.

.rodata Ta sekcija vsebuje konstante, torej podatke samo za branje. Tip sekcije je SHT_PROGBITS, aktivno ima le zastavico SHF_ALLOC.

.shstrtab To je sekcija, ki, v obliki znakovnih nizov, ločenih z ničlami, vsebuje imena vseh sekcij v datoteki ELF. Tip sekcije je SHT_STRTAB in nima aktivne nobene zastavice.

.text Ta sekcija vsebuje strojno kodo programa. Njen tip je SHT_PROGBITS, aktivni ima zastavici SHF_ALLOC in SHF_EXECINSTR.

2.3.3 Segmenti

Izvršljive datoteke ELF imajo enako zgradbo kot povezljive, podatki pa so urejeni tako, da se lahko učinkovito preslikajo v pomnilnik preden se program požene. Tako jih urejajo segmenti, ki so opisani v tabeli glav segmentov. Po tipu ločimo naložljive (tipa PT_LOAD) in ostale segmente. Naložljivi segmenti se preslikajo v pomnilnik, drugi pa označujejo pomembne dele datoteke nalagalniku. Objektna datoteka ne vsebuje veliko segmentov (navadno manj kot 10), in največkrat le dva naložljiva; enega samo za branje (ta vsebuje programsko kodo in konstante), drugega pa za branje in pisanje. Tako lahko nalagalnik segmente preslika v pomnilnik z malim številom operacij. K učinkovitosti pripomore tudi obvezna poravnava naložljivih segmentov, tako

da se odmik segmenta v datoteki ujema z navideznim naslovom segmenta v pomnilniku, po modulu velikosti pomnilniške strani. Zaradi upoštevanja te omejitve je v večini datotek ELF nekaj neizkoriščenega prostora.

Glave segmentov

Tabela glav segmentov se nahaja takoj za glavo ELF. V specifikaciji formata ELF to sicer ni predpisano, je pa s strani nekaterih operacijskih sistemov[6] (oz. njihovih nalagalnikov) določeno, da mora prva pomnilniška stran programa vsebovati poleg glave ELF tudi tabelo glav segmentov. Zato se morajo programi, ki gradijo datoteke formata ELF, tega držati. Glave segmentov (*program header*) vsebujejo podatke o segmentih v datoteki. Število glav segmentov je zapisano v glavi ELF. Nekatere opisujejo naložljive segmente, druge pa dajejo dodatne informacije, ki sicer ne prispevajo k pomnilniški sliki programa. S pomočjo glav segmentov nalagalniki iz datoteke dobijo vse potrebne podatke. Smiselne so le za izvršljive in deljene objektne datoteke.

Posamezna glava segmenta vsebuje naslednja polja:

p_type Tip segmenta. Glede na vrednost tega polja lahko interpretiramo vsebino segmenta. Nekateri tipi segmentov:

PT_LOAD Naložljiv segment. Podatki tega segmenta se iz datoteke preslikajo v pomnilnik. Če je velikost segmenta v pomnilniku večja kot velikost v datoteki, se preostanek prostora v pomnilniku zapolni z ničlami. Glave segmentov tega tipa morajo biti v tabeli glav segmentov razporejene po vrsti od tiste z najmanjšim do tiste z najvišjim navideznim naslovom v pomnilniku.

PT_DYNAMIC Tip segmenta, ki vsebuje podatke, pomembne za dinamično nalaganje.

PT_INTERP Tip segmenta, ki vsebuje pot do programa, ki naj se požene kot interpreter. Pot do interpreterja je podana z nizom

[6] To velja tudi za UNIX System V in sorodne.

znakov. Tak segment je smiseln le v izvršljivih datotekah. Pojavi se lahko le enkrat v datoteki, glava segmenta tega tipa pa mora biti v tabeli pred vsemi glavami naložljivih segmentov.

PT_PHDR Tip segmenta, ki vsebuje sámo tabelo glav segmentov. Tak segment se lahko v datoteki pojavi največ enkrat in le, če je tabela glav segmentov del pomnilniške slike programa. Glava segmenta tega tipa mora biti v tabeli pred vsemi glavami naložljivih segmentov.

p_offset Odmik od začetka datoteke do začetka podatkov segmenta.

p_vaddr Navidezni naslov v pomnilniku, kamor se bo naložil segment.

p_paddr Na sistemih, kjer je fizično naslavljanje pomnilnika smiselno, je v tem polju fizični naslov segmenta v pomnilniku. Pri UNIX System V in sorodnih sistemih je vrednost tega polja zanemarjena.

p_filesz Velikost segmenta v datoteki.

p_memsz Velikost segmenta v pomnilniku. Lahko se razlikuje od velikosti segmenta v datoteki. Če, na primer, segment vsebuje sekcijo `bss`, je velikost segmenta v pomnilniku večja od velikosti v datoteki.

p_flags To polje vsebuje zastavice, ki opisujejo segment. Nekatere možne zastavice:

PF_X Označuje izvršljiv segment.

PF_W Označuje pravico do pisanja.

PF_R Označuje pravico do branja.

p_align To število določa poravnavanje segmenta v pomnilniku in v datoteki. Za učinkovito preslikovanje naložljivih segmentov v pomnilnik, morajo biti odmiki takih segmentov v datoteki kongurentni njihovim

pomnilniškim naslovom po modulu velikosti stani. Vrednosti 0 in 1 pomenita, da poravnavanje ni potrebno, sicer pa mora veljati:

$$\mathsf{p\_align} = 2^n \quad (n \in \mathbb{N})$$
$$\mathsf{p\_vaddr} \equiv \mathsf{p\_offset} \pmod{\mathsf{p\_align}}$$

Na arhitekturi Intel x86 se za poravnavanje uporablja vrednost 1000_{16} ali več. Ker je 1000_{16} največja velikost strani za omenjeno arhitekturo, bodo datoteke, ki uporabljajo to vrednost, primerne za izvajanje na vsakem takem računalniku. Na arhitekturi x86-64 pa se uporablja vrednost 200000_{16}.

2.4 Možnosti prevedbe izvršljivih objektnih datotek med različnimi objektnimi formati

V sklopu diplomske naloge smo skušali ugotavljati tudi, kakšne so možnosti prevedbe izvršljivih datotek iz enega v drug format objektne datoteke. Namen tega bi lahko bil, na primer, izvajanje programa, napisanega za operacijski sistem Windows, v okolju Linux, ali obratno.

Sama pretvorba programske kode in drugih podatkov, shranjenih v objektnih datotekah, ni velik problem. Poznamo namreč specifikacije formatov in v skladu z njimi lahko podatke le prepišemo v drugo obliko. Za te namene že obstajajo aplikacije, na primer program objcopy, ki je del paketa GNU binutils [7]. S podobnimi aplikacijami lahko med formati uspešno pretvarjamo nekatere enostavnejše povezljive objektne datoteke. Pri izvršljivih datotekah pa hitro naletimo na celo vrsto problemov. Pri pretvorbi datoteke med objektnimi formati se lahko določeni podatki izgubijo, ker nekateri formati ne podpirajo vseh oblik podatkov drugih objektnih formatov. Na primer, posebno sekcijo `.interp` (glej opis na strani 19) pozna le format ELF. Informacije iz te sekcije v drugih objektnih formatih ne moremo zapisati.

Do težav lahko pride tudi pri zagonu programa iz pretvorjene objektne datoteke na drugem operacijskem sistemu. Nalagalnik ciljnega operacijskega sistema ima lahko drugačne zahteve kot tisti, za katerega je bila prvotna objektna datoteka namenjena. Te zahteve so lahko nalagalni naslov programa, razporeditev sekcij in segmentov, (ne)vključitev glav datoteke v pomnilniško sliko programa in podobno.

Večino teh težav bi sicer lahko odpravili z ustreznimi spremembami objektne datoteke, nov problem pa se pojavi pri dinamičnem povezovanju knjižnic, ki jih potrebuje program. Program, napisan oziroma preveden za določen operacijski sistem, pričakuje, da bo imel na voljo sistemske knjižnice, ki jih vsebuje ta operacijski sistem. Da bi odpravili ta problem, bi morali našemu programu v ciljnem operacijskem sistemu ponuditi knjižnice z enakimi funkcionalnostmi. To je mogoče, če imamo na voljo potrebne knjižnice v formatu, ki ga razume povezovalnik na novem operacijskem sistemu. Druga rešitev je, da naš program ne zahteva nobenih dodatnih knjižnic.

Največji problem prevedbe programov za delovanje na drugem operacijskem sistemu je razlika v sistemskih klicih na različnih operacijskih sistemih. S sistemskimi klici program zahteva storitve od operacijskega sistema. Ne le, da se razlikujejo številke, po katerih se identificirajo posamezni sistemski klici, tudi način proženja sistemskih klicev je lahko drugačen. Ker se sistemski klici nahajajo v sami strojni kodi programa, bi bilo njihovo iskanje in pretvorba zelo težek problem. Seveda pa ni nujno, da bi v ciljnem operacijskem sistemu za vsak sistemski klic imeli na voljo ekvivalentnega.

Kljub naštetim težavam pa lahko, s posebnimi dodatki, poganjamo nekatere programe, narejene za nek operacijski sistem, na drugem operacijskem sistemu. Program Wine v operacijskem sistemu Linux omogoča poganjanje programov, narejenih za Windows, vendar brez pretvarjanja med objektnimi formati datotek. Wine priskrbi lasten nalagalnik, ki zna delati z objektnimi datotekami formata PE. Poleg tega prispeva mnoge knjižnice, ki jih lahko programi uporabljajo kot v 'domačem' sistemu Windows. Za sistemske klice pa poskrbi s posebnim združljivostnim slojem, ki ulovi sistemske klice izva-

jajočega se programa in jih zamenja z ustreznimi sistemskimi klici za Linux. Več o programu Wine si lahko preberete v [1] in [9].

Poglavje 3

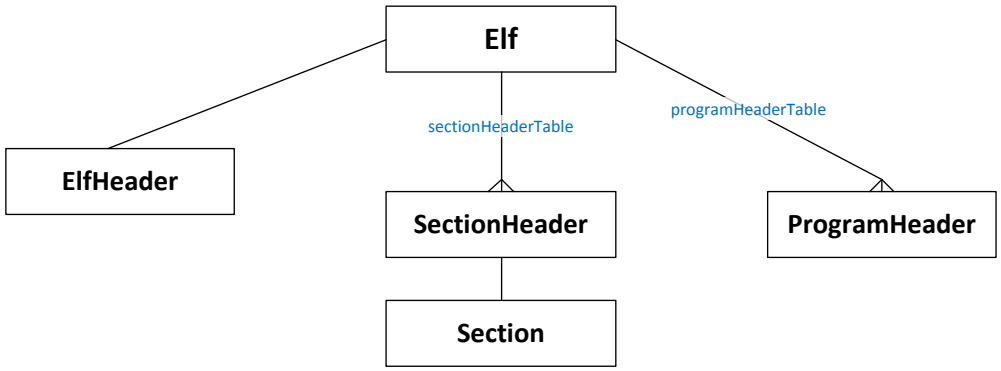
Javanski paket za delo z objektnimi datotekami ELF

Za to diplomsko nalogo je bil izdelan javanski paket, ki preslika vsebino datoteke formata ELF v javanske objekte in s tem olajša delo z datotekami ELF. Paket omogoča branje, spreminjanje, zapis in ustvarjanje nove datoteke formata ELF. V tem poglavju si bomo ogledali zgradbo in funkcionalnosti tega paketa ter probleme, ki smo jih reševali pri delu.

3.1 Zgradba

Paket je sestavljen iz razredov, ki predstavljajo dele datoteke ELF in vsebujejo polja s podatki ter metode za delo z njimi. Na sliki 3.1 so prikazani osnovni razredi in relacije med njimi. V nadaljevanju si bomo pogledali razrede, v razdelku 3.2 pa bodo predstavljene zanimivejše metode.

Razred **Elf** predstavlja celotno objektno datoteko. Objekt tega razreda vsebuje glavo ELF ter tabelo glav segmentov in tabelo glav sekcij. Razred vsebuje metode za dodajanje in odstranjevanje sekcij, izdelavo preslikave med sekcijami in segmenti, grajenje tabele imen sekcij, razporejanje sekcij in ustvarjanje nove enostavne objektne datoteke. Za dodajanje ali brisanje sekcij se namesto neposrednega spreminjanja seznama sekcij uporabljajo posebne



Slika 3.1: Razredi, ki predstavljajo datoteko ELF.

metode, ker je pri tem treba popraviti nekatera polja v glavi ELF in v sekciji, ki vsebuje imena sekcij. Razred Elf je prikazan na izpisu 3.1.

Glava ELF je predstavljena z objektom razreda **ElfHeader**. Vsebuje vsa polja v glavi datoteke ELF, razen čarobne številke, ki je implicitno določena. Vrednosti, kot so pomnilniški naslovi in odmiki posameznih delov datoteke, so po specifikaciji formata ELF predstavljene s 64-bitnimi nepredznačenimi števili. Zaradi lažjega računanja so v našem paketu taka števila predstavljena z objekti razreda java.Math.BigInteger. Največji javanski primitivni številski tip, long, namreč predstavlja 64-bitne predznačene vrednosti. Polji EI_CLASS in EI_DATA sta predstavljeni s tipom boolean, ker lahko zavzameta le dve različni vrednosti. Razred ElfHeader vsebuje tudi metode za avtomatsko nastavljanje vrednosti naslednjih polj: e_shnum, e_phnum, e_shstrndx in e_shoff. Vrednosti se pridobijo iz drugih objektov. Metoda za avtomatsko nastavljanje polja e_shoff (odmik tabele glav sekcij) postavi tabelo glav sekcij na mesto v datoteki, kjer je dovolj prostora (prostora ne zaseda nobena sekcija ali segment) oziroma na konec datoteke. Razred ElfHeader je prikazan na izpisu 3.2.

Tabela glav sekcij je predstavljena s povezanim seznamom glav sekcij. Ker ima javanski povezan seznam lahko največ $2^{31} - 1$ vnosov, je to tudi na-

```
1  public class Elf {
2      public ElfHeader elfHeader;
3      public LinkedList<ProgramHeader> programHeaderTable;
4      public LinkedList<SectionHeader> sectionHeaderTable;
5
6      public Elf(ElfHeader elfHeader){}
7      public Elf(ElfHeader elfHeader, LinkedList<ProgramHeader>
           programHeaderTable, LinkedList<SectionHeader>
           sectionHeaderTable) {}
8
9      public void removeSection(int index){}
10     public BigInteger addSection(Section section, String name,
           long sh_type, long sh_flags, BigInteger sh_addr, long
           sh_link, long sh_info, BigInteger sh_addralign,
           BigInteger sh_entsize){}
11     public void addSection(Section section, String name,
           BigInteger sh_offset, long sh_type, long sh_flags,
           BigInteger sh_addr, long sh_link, long sh_info,
           BigInteger sh_addralign, BigInteger sh_entsize){}
12     public void addSection(SectionHeader header){}
13     public void buildSectionToSegmentMapping(){}
14     public void buildShStrTab(boolean addSectionIfNotExisting)
           throws ElfException{}
15     public StrTabSection getShStrTab(){}
16     public Section getLastSection() throws ElfException{}
17     public LinkedList<ElfString> findStrings(Iterable<
           SectionHeader> shs, int minLen){}
18     public void rearrangeAllFreeSections(){}
19     public static Elf createSimpleElf(byte[] code, boolean
           buildShStrTab){}
20 }
```

Izpis 3.1: Polja in metode v razredu Elf.

```
1  public class ElfHeader {
2      public Elf elf;
3      public int e_type;
4      public int e_machine;
5      public long e_version;
6      public BigInteger e_entry;
7      public BigInteger e_phoff;
8      public BigInteger e_shoff;
9      public long e_flags;
10     public int e_ehsize;
11     public int e_phentsize;
12     public int e_phnum;
13     public int e_shentsize;
14     public int e_shnum;
15     public int e_shstrndx;
16     protected boolean ei_class64;
17     public boolean ei_dataBigEndian;
18     public int ei_version;
19     public int ei_osabi;
20     public int ei_abiversion;
21
22     public ElfHeader(Elf elf, int e_type, int e_machine, long
           e_version, BigInteger e_entry, BigInteger e_phoff,
           BigInteger e_shoff, long e_flags, int e_phnum, int
           e_shnum, int e_shstrndx, boolean ei_class64, boolean
           ei_dataBigEndian, int ei_version, int ei_osabi, int
           ei_abiversion) {}
23     public ElfHeader(InputStream in, Elf elf) throws
           ElfReaderException, IOException{}
24
25     public byte[] toBytes(){}
26     public void setE_shnum_auto(){}
27     public void setE_phnum_auto(){}
28     public void setE_shstrndx_auto(){}
29     public void setE_shoff_auto(){}
30     public void setE_shoff_auto(BigInteger restrictionStart,
           BigInteger restrictionEnd) {}
31     public void setEi_class(boolean class64){}
32     public boolean getEi_class64() {}
33 }
```

Izpis 3.2: Polja in metode v razredu ElfHeader.

jvečje število sekcij v datoteki, predstavljeni z našim paketom. Glavo sekcije predstavlja objekt razreda **SectionHeader**, ki je prikazan na izpisu 3.3. Ta vsebuje vsa polja glave sekcije ter objekt tipa Section, ki predstavlja podatke sekcije. Glava sekcije vsebuje tudi metodo za pridobivanje imena sekcije iz sekcije z imeni sekcij in metodo za branje podatkov sekcije iz datoteke, ki ustvari objekt tipa Section.

Abstraktni razred **Section** je nadrazred vsem razredom, ki predstavljajo podatke sekcije. Prikazan je v izpisu 3.5. Vsi podatki objektne datoteke, razen glav, so shranjeni v sekcijah. Datoteke s shranjenimi podatki izven sekcij v našem paketu niso podprte. Take datoteke so sicer možne, niso pa v skladu s specifikacijo formata ELF. Razred posamezne sekcije je določen glede na tip sekcije. V paketu so naslednji razredi, ki predstavljajo podatke sekcij različnih tipov:

NullSection Predstavlja sekcije tipa SHT_NULL, ki ne vsebujejo nobenih podatkov. Navadno je tega tipa le posebna sekcija na indeksu nič.

ProgbitsSection Predstavlja sekcije tipa SHT_PROGBITS. Vsebuje podatke sekcije v obliki tabele bajtov. Zaradi take predstavitve je največja velikost sekcije, ki jo paket podpira 2GB (takšna je tudi omejitev velikosti tabele v javi). Razred ProgBitsSection vsebuje metodo za iskanje nizov znakov v sekciji. Ta se sprehodi čez podatke in vrne najdene znakovne nize, daljše od izbrane minimalne dolžine. Kot znakovni niz se šteje z ničlo končano zaporedje bajtov z vrednostmi, ki ustrezajo ASCII znakom za črke in številke ter druge znake in prelome vrstic. Najdeni znakovni nizi so predstavljeni z objekti razreda **ElfString**. Za enostavnejše ravnanje z najdenimi nizi razred ElfString poleg niza znakov vsebuje tudi glavo sekcije, v kateri se niz nahaja in odmik niza od začetka sekcije.

NoBitsSection Predstavlja sekcije tipa SHT_NOBITS, ki ne vsebujejo nobenih podatkov.

StrTabSection Predstavlja sekcije tipa SHT_STRTAB, ki vsebujejo znakovne nize. Podatki so predstavljeni kot tabela bajtov, razred pa vsebuje metodo za pridobivanje niza iz izbrane lokacije v sekciji in metodo, ki iz seznama nizov sestavi podatke sekcije. Nizi znakov so v podatkih ločeni z ničlami.

DynamicSection Predstavlja sekcije tipa SHT_DYNAMIC, ki vsebujejo podatke za dinamično povezovanje. Podatki so predstavljeni kot seznam objektov razreda **DynamicStructure**.

RelSection Predstavlja sekcije tipa SHT_REL in SHT_RELA, ki vsebujejo vnose za prenaslavljanje. Podatki so predstavljeni kot seznam vnosov za prenaslavljanje, ki so objekti razreda **RelSection**.

SymTabSection Predstavlja sekcije tipa SHT_SYMTAB, ki predstavljajo simbolne tabele. Simbolna tabela je predstavljena kot seznam objektov razreda SymbolTableEntry.

NoteSection Predstavlja sekcije tipa SHT_NOTE.

DefaultSection Predstavlja sekcije drugih tipov, ki jih ne moremo predstaviti z nobenim od naštetih razredov. Podatki so predstavljeni s tabelo bajtov.

Glave segmentov so vsebovane v seznamu glav segmentov v razredu Elf. So objekti razreda **ProgramHeader**. Poleg vseh polj, ki jih vsebuje glava segmenta, ta razred vsebuje tudi seznam glav sekcij, ki, po naslovu v pomnilniku, spadajo v segment, predstavljen s to glavo. Vsebuje tudi metodo za preverjanje če naslovi segmenta ustrezajo zahtevam poravnanosti. Razred ProgramHeader je prikazan na izpisu 3.4.

Poleg razredov, ki predstavljajo dele podatkov objektne datoteke, pa naš paket vsebuje tudi razrede s statičnimi metodami za delo z javansko predstavitvijo datotek ELF. Tak je razred **ElfReader**, ki vsebuje metodo za branje datoteke ELF z diska. Ta metoda vrne objekt razreda Elf z vsemi

```
1  public class SectionHeader {
2      public Elf elf;
3      public Section section;
4      public long sh_name;
5      public long sh_type;
6      public long sh_flags;
7      public BigInteger sh_addr;
8      public BigInteger sh_offset;
9      public BigInteger sh_size;
10     public long sh_link;
11     public long sh_info;
12     public BigInteger sh_addralign;
13     public BigInteger sh_entsize;
14     public String name;
15
16     public SectionHeader(Elf elf, Section section, long sh_type,
           long sh_flags, BigInteger sh_addr, BigInteger sh_offset,
           BigInteger sh_size, long sh_link, long sh_info,
          BigInteger sh_addralign, BigInteger sh_entsize, String
          name) {}
17     public SectionHeader(InputStream in, Elf elf) throws
          ElfReaderException, IOException {}
18
19     public byte[] toBytes(){}
20     public void readSection(File file) throws
          FileNotFoundException, IOException{}
21     public String getNameFromShstrtab(){}
22     public String toString(){}
23     public void setNameFromShstrtab() {}
24     public boolean isPartOfSegment(){}
25 }
```

Izpis 3.3: Polja in metode v razredu SectionHeader.

```
1  public class ProgramHeader {
2      public Elf elf;
3      public LinkedList<SectionHeader> sections;
4      public long p_type;
5      public BigInteger p_offset;
6      public BigInteger p_vaddr;
7      public BigInteger p_paddr;
8      public BigInteger p_filesz;
9      public BigInteger p_memsz;
10     public long p_flags;
11     public BigInteger p_align;
12
13     public ProgramHeader(Elf elf, long p_type, BigInteger
           p_offset, BigInteger p_vaddr, BigInteger p_paddr,
           BigInteger p_filesz, BigInteger p_memsz, long p_flags,
           BigInteger p_align) {}
14     public ProgramHeader(InputStream in, Elf elf) throws
           ElfReaderException, IOException {}
15
16     public byte[] toBytes(){}
17     public void setP_offset(BigInteger p_offset) throws
           ElfException {}
18     public void setP_vaddr(BigInteger p_vaddr) throws
           ElfException {}
19     public void extendSize(int n) {}
20     public boolean isAlignmentOk(){}
21 }
```

Izpis 3.4: Polja in metode v razredu ProgramHeader.

```
1  public abstract class Section {
2      public SectionHeader header;
3
4      protected Section(SectionHeader header) {}
5
6      public abstract byte[] toBytes();
7      public abstract BigInteger getSize();
8      public abstract long getSh_type();
9      protected boolean setSize_auto(boolean reposition) {}
10 }
```

Izpis 3.5: Polja in metode v razredu Section.

pripadajočimi deli objektne datoteke. Pri branju datoteke se najprej preberejo glava ELF ter glave segmentov in sekcij. Po tem se za vsako sekcijo preberejo še podatki, ki se predstavijo z objektom ustreznega razreda glede na tip sekcije.

Razred **ElfWriter** vsebuje statično metodo za zapis datoteke ELF iz njene javanske predstavitve. Pred pisanjem se za vsak del datoteke ELF ustvari objekt razreda **Writable**. Ta torej lahko predstavlja glavo ELF, tabelo glav segmentov, tabelo glav sekcij ali podatke sekcije. Objekte tipa Writable lahko enostavno primerjamo po odmiku od začetka datoteke. Pred zapisom v datoteko jih zato lahko uredimo po odmikih in po vrsti zapišemo. Če si deli datoteke ne sledijo tesno, se med njimi zapišejo ničle. Do napake pa lahko pride, če se nek del datoteke začne preden se prejšnji del konča. V tem primeru metoda za zapis vrže izjemo.

Za potrebe iskanja referenc v kodi programa in za prikaz kode programa v zbirnem jeziku uporabljamo razred **Disasm**. Ta vsebuje metodo, ki, v obliki niza znakov, vrne kodo podane sekcije v zbirnem jeziku. Pri tem kliče zunanji program, povratni zbirnik (*disassembler*) ndisasm. Poda mu začasno datoteko, v kateri je strojna koda podane sekcije. Vrne izhod programa ndisasm.

V našem paketu je vsebovan tudi razred **Consts**, ki vsebuje vse po-

trebne konstante, definirane v specifikaciji formata ELF. Med drugim so to številske predstavitve tipov sekcij in segmentov, maske raznih zastavic, čarobna številka ELF in tako naprej.

3.2 Funkcionalnosti

3.2.1 Ustvarjanje nove datoteke

Poleg spreminjanja obstoječe objektne datoteke lahko s pomočjo našega paketa tudi ustvarimo novo. To lahko naredimo z uporabo konstruktorjev posameznih razredov. Datoteki moramo določiti vsaj glavo. Koda za ustvarjanje in zapis take datoteke je prikazana v izpisu 3.6. Da bi dobili smiselno datoteko, pa moramo napolniti tudi seznam glav sekcij in (opcijsko) tudi seznam glav segmentov. Glave sekcij morajo vsebovati tudi objekte razreda Section, ki predstavljajo podatke sekcij.

Enostavno izvršljivo datoteko ELF pa lahko ustvarimo tudi z uporabo metode **createSimpleElf** iz razreda Elf. Metoda kot argument sprejme programsko kodo (tabelo bajtov), ki jo želimo vključiti, vrne pa objekt razreda Elf, ki predstavlja izvršljivo datoteko. Ta ima že ustvarjeno glavo s privzetimi vrednostmi (izvršljiva datoteka, arhitektura Intel 80386...), sekcijo tipa PROGBITS ki vsebuje podano kodo ter naložljiv segment, potreben za zagon zapisanega programa.

3.2.2 Iskanje uporab naslovov

Metoda **findUsages** v razredu Disasm omogoča, da v programski kodi izbrane sekcije poiščemo ukaze, ki naslavljajo določen pomnilniški naslov. Ta funkcija je zelo koristna za proučevanje kode v objektni datoteki. Strojna koda se z uporabo povratnega zbirnika pretvori v ukaze zbirnega jezika, predstavljene v znakovnem nizu. Primer je prikazan v izpisu 3.7. V prvem stolpcu je pomnilniški naslov ukaza, v drugem stolpcu strojna koda ukaza, v tretjem stolpcu pa ukaz v zbirnem jeziku. Opazimo lahko, da nekateri ukazi vsebu-

```
ElfHeader header = new ElfHeader( /* Ustvari glavo ELF. */
  null, /* Objekt Elf, kateremu pripada ta glava. Ker ta še ni
     narejen, lahko podamo null. */
  Consts.ET_EXEC, /* Tip objektne datoteke. */
  Consts.EM_X86_64, /* Arhitektura ciljnega računalnika. */
  Consts.EV_CURRENT, /* Verzija formata datoteke. */
  BigInteger.ZERO, /* Vstopna točka programa. */
  BigInteger.ZERO, /* Lokacija tabele glav segmentov. */
  BigInteger.ZERO, /* Lokacija tabele glav sekcij. */
  0, /* Zastavice. */
  0, /* Število glav segmentov. */
  0, /* Število glav sekcij. */
  Consts.SHN_UNDEF, /* Indeks sekcije z imeni sekcij. */
  true, /* Razred datoteke (64-bitna). */
  false, /* Kodiranje operandov (pravilo tankega konca). */
  Consts.EV_CURRENT, /* Verzija formata identifikacije. */
  0, /* Identifikacija operacijskega sistema. */
  0 /* Verzija identifikacije operacijskega sistema. */
);
Elf e = new Elf(header); /* Ustvari objekt Elf. */
ElfWriter.save(e, new File("output")); /* Zapis v datoteko z
   imenom "output". */
```

Izpis 3.6: Ustvarjanje javanske predstavitve datoteke ELF brez vsebine.

jejo naslove, na katere se sklicujejo. Pri dveh ukazih se pred naslovom pojavi besedica 'rel', ki označuje, da je tukaj uporabljeno PC-relativno naslavljanje. To lahko opazimo tudi s pregledom strojne kode ukaza. Poglejmo tretji ukaz v izpisu. Ta naslavlja naslov 601028_{16}. Strojna koda ukaza pa namesto te vrednosti vsebuje $200B68_{16}$, kar je razlika med naslovljenim naslovom in naslovom naslednjega ukaza (kamor ob izvajanju kaže programski števec). Ker je uporabljeno pravilo tankega konca, število najdemo v kodi zapisano kot 680B20.

S pomočjo regularnega izraza[1] se v zbirnem jeziku poiščejo vrstice, ki

[1]Uporablja se paket [3] za regularne izraze.

vsebujejo izbran naslov. Za vsako od teh vrstic se ustvari objekt razreda **ElfReference**. Tak objekt vsebuje ukaz v zbirnem jeziku, njegov naslov v datoteki, naslov reference na naslov znotraj ukaza, naslov ukaza v pomnilniku, dolžino ukaza v bajtih in informacijo, ali je bilo pri ukazu uporabljeno PC-relativno naslavljanje. Te podatke potrebujemo, če želimo v programski kodi zamenjati naslov, na katerega se ukaz sklicuje. Do napak bi lahko prišlo, če bi namesto naslova v ukazu zaznali konstanto z enako vrednostjo. Tega naš enostaven iskalnik sklicev ne bi zaznal nikakor drugače. Zato je priporočljivo, da pred programskim spreminjanjem kode pravilnost najdenih referenc preveri uporabnik.

```
1  004004B4  53              push rbx
2  004004B5  4883EC08        sub rsp,byte +0x8
3  004004B9  803D680B200000  cmp byte [rel 0x601028],0x0
4  004004C0  754B            jnz 0x40050d
5  004004C2  BB400E6000      mov ebx,0x600e40
6  004004C7  488B05620B2000  mov rax,[rel 0x601030]
7  004004CE  4881EB380E6000  sub rbx,0x600e38
8  004004D5  48C1FB03        sar rbx,0x3
9  004004D9  4883EB01        sub rbx,byte +0x1
10 004004DD  4839D8          cmp rax,rbx
11 004004E0  7324            jnc 0x400506
```

Izpis 3.7: Del kode v zbirnem jeziku, kot jo dobimo iz povratnega zbirnika.

3.2.3 Razporeditev sekcij znotraj datoteke

Pri dodajanju ali povečevanju sekcij lahko hitro naletimo na prekrivanje sekcij v objektni datoteki. Tudi če dodajamo sekcijo na konec datoteke, se pri tem poveča tabela glav sekcij in pri tem lahko prekrije del naslednje sekcije. Zato nam pride prav sposobnost avtomatskega razporejanja sekcij. Na ta način lahko razporejamo tabelo glav sekcij in sekcije, ki niso del nobenega segmenta.

Če bi prestavljali sekcije, ki so del segmentov, bi jih s tem premaknili tudi v pomnilniški sliki programa, s tem pa bi pokvarili vsa sklicevanja na naslove

znotraj premaknjene sekcije. V tem primeru bi morali popraviti vse ukaze, ki se sklicujejo na katerikoli naslov znotraj premaknjene sekcije. To bi bilo v določenih primerih sicer možno, vendar se s tem naš paket ne ukvarja.

Metoda **rearrangeAllFreeSections** iz razreda Elf prestavi vsako sekcijo, ki ni del segmenta, na prvo mesto v datoteki, kjer je dovolj prostora. Pri tem sekcij ne postavlja v prostor, ki pripada kateremu od segmentov.

Poglavje 4

Grafični vmesnik za delo z objektnimi datotekami ELF

Razvit paket, opisan v prejšnjem poglavju, smo uporabili za izdelavo programa z grafičnim vmesnikom za delo z objektnimi datotekami formata ELF. Namenjen je predvsem pregledu in spoznavanju objektnih datotek, omogoča pa tudi kreiranje nove enostavne datoteke in zamenjavo nizov, uporabljenih v programu. V tem poglavju si bomo pogledali funkcije in izgled tega programa.

4.1 Ustvarjanje enostavne izvršljive datoteke

Poleg odpiranja obstoječe datoteke ELF je na voljo tudi funkcija za ustvarjanje nove enostavne izvršljive ELF datoteke, ki smo jo opisali v razdelku 3.2.1. Za ustvarjanje nove datoteke mora uporabnik vnesti strojno kodo programa s šestnajstiškimi števili. Ta funkcija je uporabna le za ustvarjanje najenostavnejših izvršljivih datotek s programom, ki ga že imamo v obliki strojne kode. Na ta način lahko dobimo zelo majhne izvršljive datoteke, kakršnih z običajnimi povezovalniki ne moremo, saj ti v datoteke vključujejo več sekcij.

Na izpisu 4.1 si poglejmo kodo kratkega programa za procesor arhitekture

```
1 31C0              xor eax,eax
2 40                inc eax
3 BB2A000000        mov ebx,42
4 CD80              int 0x80
```

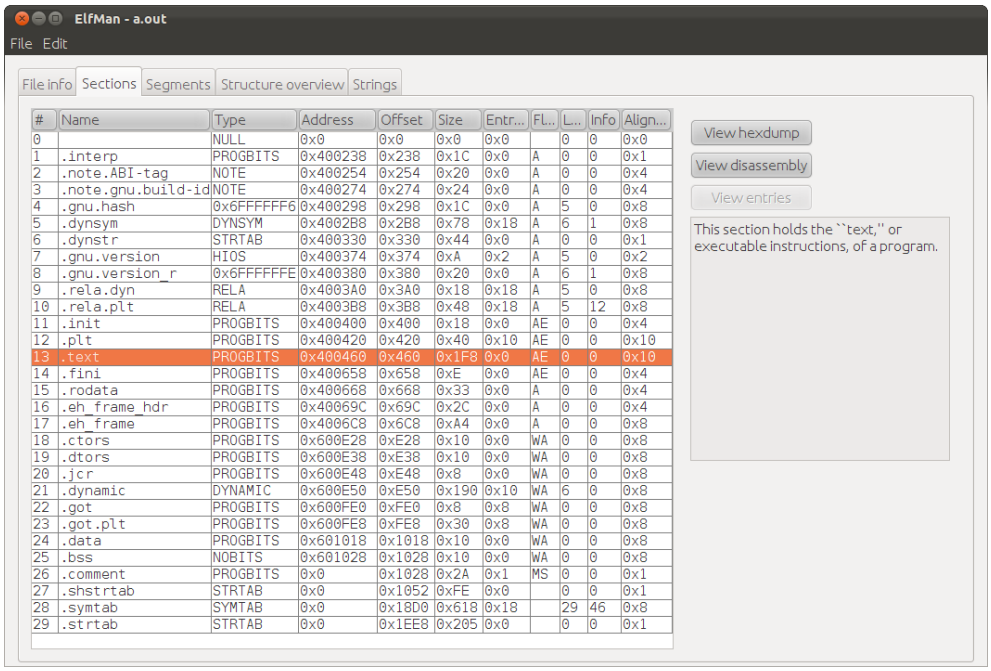
Izpis 4.1: Primer zelo kratkega programa.

Intel 80386 in okolje Linux. Njegova naloga je le izhod s kodo izhoda 42. Za izhod mora program sprožiti ustrezen sistemski klic. Za vse sistemske klice se v operacijskem sistemu Linux sproži prekinitev 80_{16} (ukaz 4). Medtem se mora v registru eax nahajati številka sistemskega klica za izhod, to je 1, v registru ebx pa koda izhoda. Kot lahko opazimo iz izpisa, smo za nalaganje vrednosti 1 v register eax uporabili dva ukaza, xor (ekskluzivna disjunkcija registra s samim sabo) in inc (povečanje vrednosti registra za 1). Ta ukaza skupaj tvorita krajšo strojno kodo kot en sam ukaz mov za takojšnje nalaganje vrednosti v register. Tak ukaz pa smo uporabili za nalaganje vrednosti 42 v register ebx.

S pomočjo zbirnika smo ukaze pretvorili v strojno kodo, za izdelavo izvršljive objektne datoteke pa smo uporabili naš program. Strojna koda programa je skupaj dolga le 10 bajtov. Poleg kode pa objektna datoteka vsebuje še glavo ELF (52 bajtov), eno glavo segmenta (32 bajtov) in eno glavo sekcije (40 bajtov). To znese skupaj 134 bajtov, datoteka programa z enako funkcijo, napisanega v jeziku C ter prevedenega s prevajalnikom GCC (s privzetimi nastavitvami) pa zasede 7122 bajtov. V praksi bi lahko ustvarili še manjšo delujočo izvršljivo datoteko z enakim programom (dolgo le 45 bajtov), vendar bi ta kršila specifikacije formata ELF. Postopek je na zanimiv način opisan v [6].

4.2 Pregled objektne datoteke

Pregled objektne datoteke je po zavihkih razdeljen na pet delov: informacije o datoteki, sekcije, segmenti, pregled sestave datoteke in iskalnik nizov.



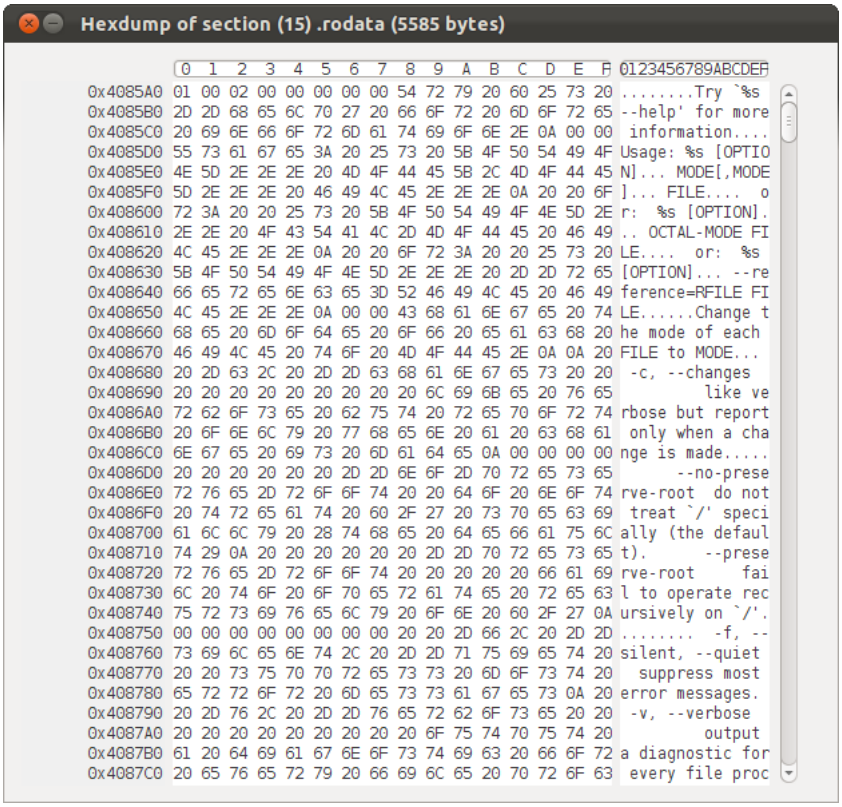
#	Name	Type	Address	Offset	Size	Entr...	Fl...	L...	Info	Align...
0		NULL	0x0	0x0	0x0	0x0		0	0	0x0
1	.interp	PROGBITS	0x400238	0x238	0x1C	0x0	A	0	0	0x1
2	.note.ABI-tag	NOTE	0x400254	0x254	0x20	0x0	A	0	0	0x4
3	.note.gnu.build-id	NOTE	0x400274	0x274	0x24	0x0	A	0	0	0x4
4	.gnu.hash	0x6FFFFFF6	0x400298	0x298	0x1C	0x0	A	5	0	0x8
5	.dynsym	DYNSYM	0x4002B8	0x2B8	0x78	0x18	A	6	1	0x8
6	.dynstr	STRTAB	0x400330	0x330	0x44	0x0	A	0	0	0x1
7	.gnu.version	HIOS	0x400374	0x374	0xA	0x2	A	5	0	0x2
8	.gnu.version_r	0x6FFFFFFE	0x400380	0x380	0x20	0x0	A	6	1	0x8
9	.rela.dyn	RELA	0x4003A0	0x3A0	0x18	0x18	A	5	0	0x8
10	.rela.plt	RELA	0x4003B8	0x3B8	0x48	0x18	A	5	12	0x8
11	.init	PROGBITS	0x400400	0x400	0x18	0x0	AE	0	0	0x4
12	.plt	PROGBITS	0x400420	0x420	0x40	0x10	AE	0	0	0x10
13	.text	PROGBITS	0x400460	0x460	0x1F8	0x0	AE	0	0	0x10
14	.fini	PROGBITS	0x400658	0x658	0xE	0x0	AE	0	0	0x4
15	.rodata	PROGBITS	0x400668	0x668	0x33	0x0	A	0	0	0x4
16	.eh_frame_hdr	PROGBITS	0x40069C	0x69C	0x2C	0x0	A	0	0	0x4
17	.eh_frame	PROGBITS	0x4006C8	0x6C8	0xA4	0x0	A	0	0	0x8
18	.ctors	PROGBITS	0x600E28	0xE28	0x10	0x0	WA	0	0	0x8
19	.dtors	PROGBITS	0x600E38	0xE38	0x10	0x0	WA	0	0	0x8
20	.jcr	PROGBITS	0x600E48	0xE48	0x8	0x0	WA	0	0	0x8
21	.dynamic	DYNAMIC	0x600E50	0xE50	0x190	0x10	WA	6	0	0x8
22	.got	PROGBITS	0x600FE0	0xFE0	0x8	0x8	WA	0	0	0x8
23	.got.plt	PROGBITS	0x600FE8	0xFE8	0x30	0x8	WA	0	0	0x8
24	.data	PROGBITS	0x601018	0x1018	0x10	0x0	WA	0	0	0x8
25	.bss	NOBITS	0x601028	0x1028	0x10	0x0	WA	0	0	0x8
26	.comment	PROGBITS	0x0	0x1028	0x2A	0x1	MS	0	0	0x1
27	.shstrtab	STRTAB	0x0	0x1052	0xFE	0x0		0	0	0x1
28	.symtab	SYMTAB	0x0	0x18D0	0x618	0x18		29	46	0x8
29	.strtab	STRTAB	0x0	0x1EE8	0x205	0x0		0	0	0x1

Slika 4.1: Zaslonski posnetek pregleda sekcij.

Zavihek z informacijami o datoteki prikazuje podatke, ki so zapisani v glavi datoteke.

Pregled sekcij je prikazan na sliki 4.1 ter vsebuje tabelo z vsemi sekcijami in podatki iz njihovih glav. Prikazano objektno datoteko je ustvaril prevajalnik GCC iz programa, napisanega v programskem jeziku C. Označena je sekcija .text. Ker je to ime sekcije rezervirano po specifikaciji formata ELF, je v okvirčku na desni prikazan opis te sekcije. Z gumbi nad tem okvirčkom lahko uporabnik pri vsaki sekciji, ki vsebuje podatke, izbere šestnajstiški prikaz podatkov (prikazan na sliki 4.2) ali prikaz obratnega zbirnika. Obratni zbirnik iz vseh podatkov sekcije izpiše kodo v zbirnem jeziku, ki pa je seveda smiselna le za tiste sekcije, ki vsebujejo strojno kodo. Pri sekcijah, ki vsebujejo tabele vnosov, lahko izberemo tudi prikaz vnosov. Ta prikaže tabelo vnosov v berljivi obliki.

Podobno kot pregled sekcij tudi pregled segmentov prikazuje tabelo s podatki iz glav segmentov. V tabeli se ob podatku o obvezni poravnanosti



Slika 4.2: Zaslonski posnetek šestnajstiškega prikaza sekcije `.rodata` programa chmod. Na levi strani so prikazani pomnilniški naslovi posameznih vrstic, na desni pa znakovna predstavitev podatkov s kodiranjem ASCII.

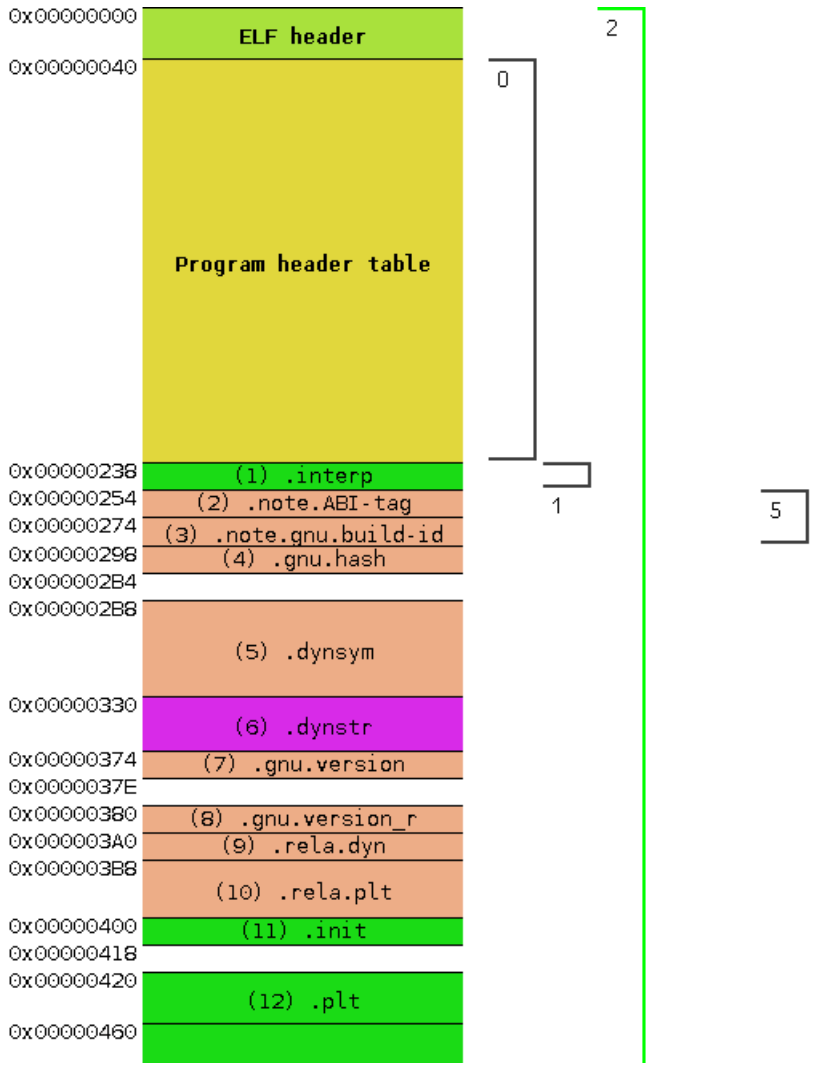
segmenta izpiše klicaj, če zahteva po poravnanosti ni izpolnjena. Ob izboru posameznega segmenta se ob strani izpiše spisek sekcij, ki spadajo v pomnilniško sliko izbranega segmenta.

Pregled sestave datoteke prikazuje fizično zgradbo datoteke in razmerja med velikostmi posameznih delov. S pravokotniki so prikazani vsi deli datoteke, to so glava ELF, tabela glav segmentov, tabela glav sekcij, posamezne sekcije in tudi morebiten neizkoriščen prostor. Na posameznem pravokotniku je označeno, kateri del datoteke predstavlja, njegova višina pa je sorazmerna velikosti tega dela datoteke. Če je del datoteke prevelik, pa je zavoljo preglednosti narisan posebej označen nesorazmeren pravokotnik. Ob vsakem delu datoteke je zapisan njegov odmik od začetka datoteke. Če se dela datoteke prekrivata, je odmik označen z rdečo barvo. Ob strani je prikazan tudi obseg posameznih segmentov, pri čemer so posebej označeni naložljivi segmenti. Del pregleda sestave datoteke je prikazan na sliki 4.3.

V naš program je vključen tudi iskalnik nizov, ki med podatki sekcij tipa SHT_PROGBITS poišče znakovne nize. Uporabnik lahko v zavihku z iskalnikom nizov izbere sekcije, v katerih naj se nizi iščejo, minimalno dolžino najdenega niza, lahko pa tudi išče po najdenih nizih. Poleg vsakega najdenega niza so prikazani tudi podatki o tem, kje se nahaja v datoteki in kje v pomnilniški sliki programa.

4.3 Urejanje nizov v izvršljivih objektnih datotekah

Funkcija urejanja nizov je dosegljiva iz zavihka iskalnika nizov. Omogoča nam urejanje znakovnih nizov v programski kodi izvršljivih objektnih datotek formata ELF. Urejanje nizov v povezljivih datotekah bi zahtevalo nekoliko drugačen pristop, s tem pa se v tej nalogi ne bomo ukvarjali. V nadaljevanju je opisano delovanje te funkcije, v praksi pa jo lahko izkoristimo na primer za prevajanje uporabniških vmesnikov programov, za katere nimamo izvorne kode.



Slika 4.3: Del pregleda strukture objektne datoteke. Uporabljena je enaka datoteka kot na sliki 4.1.

4.3.1 Shranjevanje in naslavljanje nizov

Programi seveda lahko hranijo in naslavljajo znakovne nize na nešteto načinov, lahko bi jih celo izračunavali med izvajanjem. Zato ni mogoče v vsakem primeru najti ali uspešno urediti niza, ki ga program uporablja. Večina programov (prevajalnikov) pa se drži določenih pravil, ki jih lahko izkoristimo. Znakovni nizi, ki se pojavljajo v uporabniškem vmesniku programa navadno spadajo med podatke samo za branje, zato so shranjeni v sekciji `.rodata`. Velja tudi pravilo, da se končajo z znakom 0 in večina programov je narejenih tako, da pri izpisovanju izpisujejo znake niza po vrsti dokler ne naletijo na znak 0. Poseben primer so programi, napisani v programskem jeziku Pascal. Ti hranijo dolžino niza v bajtu pred prvim znakom, nizi pa se tudi v takih programih zaključijo z znakom 0.

Nizi so običajno naslovljeni na njihov prvi znak, lahko pa bi program naslavljal niz tudi na sredini. Če bi podatki programa vsebovali niz "model" in bi program uporabljal besedi "model" in "del", bi lahko v prvem primeru naslovil prvi znak niza, v drugem primeru pa tretji znak. Če bi takemu programu spremenili niz "model", bi mu seveda pokvarili tudi besedo "del". Tudi zaradi tega moramo biti pri urejanju nizov v programih previdni, čeprav prevajalniki tudi v takih primerih ponavadi shranijo oba niza posebej.

4.3.2 Shranjevanje novega niza

Urejanje niza znotraj objektne datoteke je trivialna operacija, če je dolžina novega manjša ali enaka dolžini prvotnega niza. V tem primeru se niz le prepiše, na konec novega niza pa se vstavi znak 0. Če imamo opravka s prej omenjenim nizom programskega jezika Pascal, moramo popraviti tudi bajt pred nizom, ki označuje njegovo dolžino. Ker se naslov niza ohrani, programske kode ni potrebno spreminjati.

Če pa je nov niz daljši od originala, pa ga ne moremo zapisati na isto mesto. V tem primeru se v datoteki ustvari nova sekcija, kamor se zapiše nov niz, ali pa razširi obstoječa sekcija, ki vsebuje spremenjene nize. Da

se bo nov niz tudi naložil v pomnilniško sliko programa, moramo to sekcijo vključiti v enega od naložljivih segmentov ali ustvariti nov segment. Če imamo v datoteki na koncu obstoječega segmenta dovolj prostora, lahko novo sekcijo vključimo na to mesto in razširimo segment za dolžino nove sekcije. Dovolj prostora na koncu segmenta pomeni, da je v datoteki na tistem mestu bodisi neizkoriščen prostor bodisi sekcije, ki niso del segmentov in jih lahko prestavimo.

Če za nobenim obstoječim naložljivim segmentom ni dovolj prostora, ali če nočemo razširiti segmenta v katerem je dovoljeno pisanje, moramo za novo sekcijo ustvariti nov segment. Ustvarjanje novega segmenta v obstoječi izvršljivi datoteki ni trivialno, ker običajno nimamo prostora za razširitev tabele glav segmentov, ki mora zaradi zahtev nalagalnika ostati na svojem mestu. Lahko pa novo glavo segmenta zapišemo na mesto obstoječe in s tem povozimo obstoječ segment. Seveda ne smemo tako povoziti kateregakoli segmenta, na srečo pa ima večina izvršljivih datotek tudi segment, ki ga za izvajanje ne potrebuje. To je segment tipa PT_NOTE, ki vsebuje le informacije o prevajalniku, ki je ustvaril datoteko. Novo glavo segmenta lahko torej zapišemo na mesto glave segmenta tipa PT_NOTE in s tem ne pokvarimo delovanja programa. Nov naložljiv segment se mora v pomnilniku nahajati na lastni strani, zato bo po tem program zasedel nekoliko več prostora v pomnilniku. Pri vstavljanju nove sekcije in segmenta v datoteko moramo paziti tudi na poravnanost pomnilniškega naslova segmenta z njegovim odmikom v datoteki.

4.3.3 Spreminjanje programske kode

Če smo v datoteko vstavili niz na novo mesto, moramo popraviti tudi programsko kodo, da bo namesto starega naslovljen novi niz. Za to uporabimo funkcijo za iskanje ukazov, opisano v razdelku 3.2.2. Iščemo ukaze, ki naslavljajo prvi znak[1] starega niza. V kodi najdenih ukazov moramo spre-

[1] Če ima niz shranjeno dolžino v bajtu pred prvim znakom, moramo iskati naslov tega bajta, saj program niz naslavlja tam.

meniti naslov na naslov novega niza. Pri tem moramo upoštevati morebitno uporabo relativnega naslavljanja.

4.3.4 Uspešnost metode

Naš pristop k urejanju nizov se zanaša na že omenjene predpostavke, da se program drži določenih pravil o shranjevanju in naslavljanju nizov. Pri nekaterih programih, predvsem tistih z grafičnim vmesnikom, pa lahko naletimo na drugačno obliko shranjevanja nizov. Programi lahko preko dinamičnega povezovalnika uporabljajo nize tudi iz zunanjih knjižnic. Težavo predstavljajo tudi nizi v drugih oblikah kodiranja, na primer Unicode. Prilagoditev programa za druga kodiranja bi bila možna, vendar se s tem v tej nalogi nismo ukvarjali. Čeprav je podprto urejanje omenjenih nizov, ki hranijo njihovo dolžino, pa niso podprti taki nizi, daljši od 255 znakov. V tem primeru dolžina niza presega število, ki ga lahko shranimo v en bajt in se shrani kot 32-bitno število.

Kljub naštetim možnostim za težave pa smo na večini preizkušenih programov lahko zamenjali vsaj nekaj nizov, ki so del uporabniškega vmesnika. Primer izvajanja uspešno spremenjenega programa je prikazan na izpisu 4.2. Niz "*No such file or directory*" ni del programa, temveč ga kot sporočilo o napaki posreduje jedro operacijskega sistema.

```
anze@anze-linux:~$ chmod
chmod: missing operand
Try 'chmod --help' for more information.

anze@anze-linux:~$ chmod +x a
chmod: cannot access 'a': No such file or directory

anze@anze-linux:~$ ./chmod
./chmod: manjkajoc operand
Poskusi './chmod --help' za vec informacij.

anze@anze-linux:~$ ./chmod +x a
./chmod: ne morem dostopati do 'a': No such file or directory
```

Izpis 4.2: Primerjava izvajanja originalnega programa chmod s programom, v katerem smo spremenili nekatere nize.

Poglavje 5

Sklepne ugotovitve

V okviru te diplomske naloge smo spoznali različne formate objektnih datotek ter razvili orodje za delo z datotekami formata ELF v obliki javanskega paketa in samostojnega programa z grafičnim vmesnikom. Javanski paket je namenjen splošnemu delu z objektnimi datotekami, glavni prispevek samostojnega programa pa je funkcija urejanja nizov v izvršljivih programih. Na internetu smo sicer našli obstoječe knjižnice, podobne našemu paketu, vendar ne za programski jezik Java in brez nekaterih funkcionalnosti našega paketa kot sta iskanje referenc na naslov v programski kodi in avtomatsko razporejanje sekcij. Zasledili smo tudi grafične prikazovalnike strukture datotek ELF in enostavne urejevalnike nizov v programih, vendar ne takih, ki bi omogočali vstavljanje nizov, daljših od prvotnega.

Razvit paket se lahko uporablja za analizo objektnih datotek, za spreminjanje objektnih datotek v razne namene ali za izdelavo objektnih datotek v prevajalnikih in povezovalnikih. Grafično orodje pa je namenjeno spoznavanju zgradbe objektnih datotek ter primerjavi različnih prevajalnikov in povezovalnikov. Funkcija urejanja nizov je uporabna za prevajanje uporabniških vmesnikov izvedljivih programov.

Možnosti za izboljšave in nadgradnje so tako pri javanskem paketu kot pri grafičnem vmesniku velike. Paket bi lahko razširili, da bi omogočal delo tudi z drugimi formati objektnih datotek ter pretvarjanje med njimi. Kot

smo opisali v razdelku 2.4, bi bila uporabnost tega vprašljiva, poleg tega pa orodja za pretvarjanje med formati objektnih datotek že obstajajo. Grafični program bi lahko nadgradili tako, da bi omogočal urejanje vseh podrobnosti datotek ELF in izdelavo objektnih datotek po meri.

Literatura

[1] B. Amstadt in M. K. Johnson, "Wine", v zborniku: Linux Journal, avgust 1994. Dostopno na:
http://www.linuxjournal.com/article/2788

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

[3] Javanski paket *dk.brics.automaton*. A. Møller, Aarhus University, 2011. Dostopno na:
http://www.brics.dk/automaton/

[4] M. Pietrek, "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format", v zborniku: Microsoft Systems Journal (ur. M. Freeman), marec 1994. Dostopno na:
http://msdn.microsoft.com/en-us/magazine/ms809762.aspx

[5] M. Pietrek, "An In-Depth Look into the Win32 Portable Executable File Format", v zborniku: MSDN Magazine, februar 2002. Dostopno na:
http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

[6] B. Raiter, "A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux". Dostopno na:
http://www.muppetlabs.com/ breadbox/software/tiny/teensy.html

[7] GNU binutils, Free Software Foundation, Inc, 2007. Dostopno na:
http://www.gnu.org/software/binutils/

[8] *System V Application Binary Interface*. SCO Group. Dostopno na: http://www.sco.com/developers/gabi/latest/contents.html

[9] (2012) Wine Wiki, The Wine Project. Dostopno na: http://wiki.winehq.org/

[10] *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, Version 1.2, 1995. Dostopno na: http://refspecs.freestandards.org/elf/elf.pdf

[11] *ELF-64 Object File Format*, Version 1.5, 1998. Dostopno na: http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf