VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TRANSFORMACE WINDOWS PE DO GRAFU TOKU ŘÍZENÍ

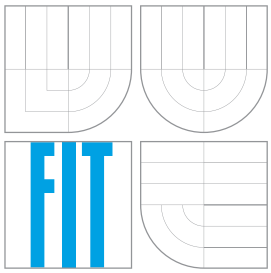
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. OTA JIRÁK

BRNO 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TRANSFORMACE WINDOWS PE DO GRAFU TOKU ŘÍZENÍ

WINDOWS PE TRANSFORMATION INTO CONTROL FLOW GRAPH

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. OTA JIRÁK

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Dr. Ing. DUŠAN KOLÁŘ

BRNO 2007

Zadání diplomové práce/4717/2006/xjirak03

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání diplomové práce

Řešitel: **Jirák Ota, Bc.**
Obor: Informační systémy
Téma: **Transformace Windows PE do grafu toku řízení**
Kategorie: Překladače

Pokyny:

1. Prostudujte formát Windows Program Executable (PE).
2. Prostudujte instrukční sadu Intelx86.
3. Implementujte analyzátor PE poskytující textový formát instrukcí.
4. Implementuje analýzu toku řízení (sestavení grafu toku řízení) a její uložení vhodnou formou do souboru pro další zpracování.
5. Ověřte na sadě vhodných příkladů správnou funkci programu.
6. Zhodnoťte přínos své práce, diskutujte možná rozšíření či případné nedostatky své práce.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části diplomového projektu je požadováno:

- První 3 body zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kolář Dušan, doc. Dr. Ing.**, UIFS FIT VUT
Datum zadání: 28. února 2006
Datum odevzdání: 22. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Bc. Ota Jirák**
Id studenta: 49386
Bytem: Sázavská 584, 582 91 Světlá nad Sázavou
Narozen: 12. 02. 1983, Havlíčkův Brod
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
diplomová práce

Název VŠKP: Transformace Windows PE do grafu toku řízení
Vedoucí/školitel VŠKP: Kolář Dušan, doc. Dr. Ing.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☒ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

Abstrakt

Tato práce pojednává o formátu spustitelných souborů EXE. Soustředí se na části potřebné při reverzním inženýrství. Dále se zabývá jazykem symbolických instrukcí, jeho reprezentací v binárních souborech a získání textového popisu instrukcí disasemblerem. Následuje popis převodu na graf toku řízení, detekce základních struktur (větvení a cykly) vyšších programovacích jazyků.

Klíčová slova

EXE, PE, COFF, formát programu, formát instrukcí asembleru, asembler, disasembler, graf toku řízení,analýza grafu

Abstract

This thesis is interested in format of executable files EXE. It is focused on parts relevant for reverse engineering. It is interested in assembler, binary representation of instruction and disassembling. Follow I introduce converting from executables to control flow graph, basic structures (branches, cycles) detection.

Keywords

EXE, PE, COFF, format of programs, format of assembler instructions, assembler, disassembler, control flow graph, graph analyzing

Citace

Ota Jirák: Transformace Windows PE do grafu toku řízení, diplomová práce, Brno, FIT VUT v Brně, 2007

Transformace Windows PE do grafu toku řízení

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doc. Dr. Ing. Dušana Koláře. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ota Jirák
21. května 2007

Obsah

Kapitola 1

Základní pojmy

Tento projekt je ve své podstatě úlohou reverzního inženýrství. Úkolem je z binárního souboru získat textovou podobu instrukcí. K tomu slouží nástroje zvané disasemblery. Existuje řada různě kvalitních nástrojů poskytující tuto funkčnost. Např. OllyDbg, dumpbin. K vytvoření vlastního disasembleru vede jeho využití při analýze spustitelných programů. Při čtení následujících kapitol narazíme na několik pojmů, které bychom si měli nyní objasnit.

- relativní adresa - jedná se o vyjádření vzdálenosti od současné adresy.
- virtuální prostor - virtuální paměťový prostor. Virtuální adresa určitého místa v tomto prostoru je vždy stejná. Nezávisí na fyzickém umístění v paměti.
- relativní virtuální adresa - zkráceně RVA. Je to to samé, jako relativní adresa, ale ve virtuálním prostoru adres.
- virtuální adresa - VA je adresou ve virtuálním prostoru.
- posunutí - posun v paměti. Využívá se při adresování paměti.
- přímá data - přímo vložená hodnota do instrukce.
- opkód - operační kód, kód instrukce.
- booleova podmínka - výraz logické aritmetiky, výstupem je true (pravda) nebo false (nepravda).
- callback funkce - funkce, která se zavolá při nějaké události. Ukazatel na tuto funkci je předán jako parametr jiné funkce.
- řídké pole - tabulka s malou obsazeností položek.

Kapitola 2

Úvod

S rozvojem internetu jsou možnosti šíření škodlivého softwaru stále větší. Nebezpečí se může skrývat v podobě přílohy elektronické pošty, makra dokumentu, či na první pohled neškodného programu, poskytujícího nějaké služby. V ohrožení nejsou jen velké firemní sítě, ale i domácí pracovní stanice. Proto jsou v dnešní době programy zajišťující bezpečnost počítačových systémů nutností.

Škodlivý software se nejčastěji objevuje ve formě spustitelného programu EXE na platformě Microsoft Windows. Tématem této práce je proto seznámení se s jejich formátem, tj. Microsoft Windows Portable Executable File Format. Tuto formu má běžný EXE soubor, dynamicky linkovaná knihovna DLL a řada dalších typů souborů. Zaměříme se na 32-bitovou architekturu.

Práce je rozdělena na prerekvizity, analýza a implementace. Zhodnocení práce se nachází v závěru. Třetí kapitola (prerekvizity) seznamuje se strukturou souborů EXE. Zaměříme se především na části potřebné pro vývoj disasembleru a analýzu kódu. Nejprve se seznámíme s několika základními pojmy. Jejich znalost je nutná k pochopení následujících částí. V druhé části této kapitoly se zaměříme na formát instukcí Intel x86. Třetí část kapitoly se zaměřuje na problematiku grafu toku řízení. Definice grafů, jejich získávání a možnosti využití.

Čtvrtá kapitola obsahuje analýzu problematiky tvorby disasembleru a získání grafu toku řízení. Také se zaměří na detekování cyklů a větvení vyšších programovacích jazyků.

V kapitole páté jsou probrány důležité části implementace.

Tato diplomová práce navazuje na semestrální projekt, ve kterém byla zpracována problematika PE formátu, binárního vyjádření jazyka symbolických instrukcí a tvorba disasembleru. Výsledky této práce jsou použity v kapitolách 3 až 5.

Kapitola 3

Prerekvizity

3.1 PE formát

Microsoft *Portable Executable File Format* [6], známý také jako PE (Portable Executable) formát, je součástí původní specifikace Win32[1]. Tento návrh je odvozen z dřívějšího formátu *Common Object File Format* (COFF) využívaného na platformě VAX/VMS. Jak již samotný název napovídá, tato specifikace se snaží o vytvoření souborového formátu přenosného mezi distribucemi operačního systému Microsoft Windows. Jedná se o Windows NT, Windows 95 a jejich následníky. Stejný formát je použit i v distribuci Windows CE.

S nástupem 64-bitových technologií vznikl nový formát označovaný jako PE 32+. Rozdíly mezi formáty PE a PE 32+ jsou nepatrné. V hlavičkách 64-bitových souborů nebyla přidána nová pole. Jedno pole bylo odstraněno a několik jich bylo rozšířeno z 32 na 64 bitů. Tento návrh umožňuje psaní programů fungujících v 32-bitových i 64-bitových PE souborech.

Stejného formátu využívají jak spustitelné soubory (EXE), tak dynamicky linkované knihovny (DLL). Rozdíl je jen v nastavení příznaku v hlavičce souboru a ve způsobu jejich používání. Soubory EXE jsou spouštěny přímo. Programy (EXE soubory) při vykonávání své funkčnosti využívají funkce obsažené v DLL.

Dynamické knihovny nejsou jenom soubory s příponou DLL. Jedná se také o soubory s příponou OCX, CPL, DRV.

Hlavička PE souboru obsahuje tyto části: MS-DOS, signaturu PE a hlavičky nazývané Souborová hlavička (angl. *FileHeader*) a Volitelná hlavička (angl. *OptionalHeader*) (viz obr. 3.1).

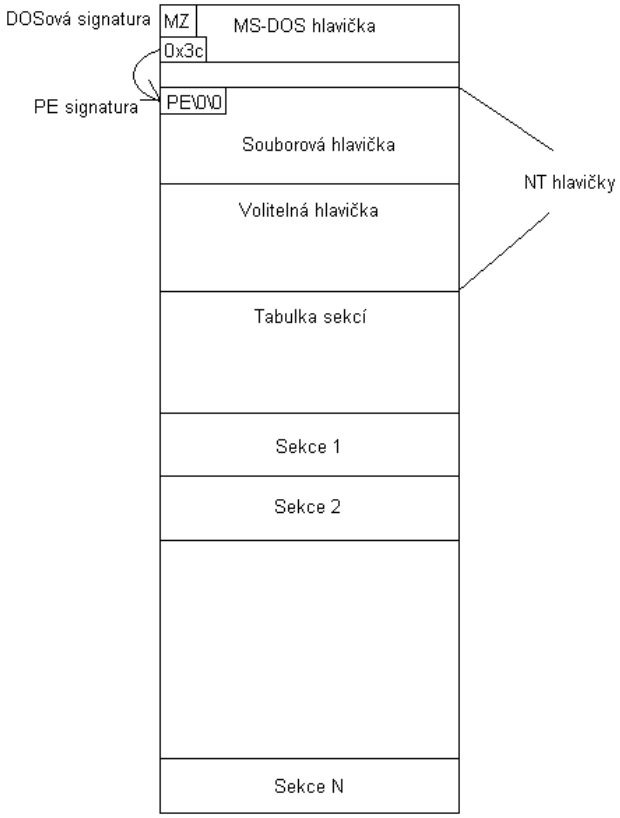
3.1.1 MS-DOS

Část souboru, v originále zvaná "MS-DOS Stub", je aplikací MS-DOS umístěnou na začátku každého EXE souboru. Při spuštění programu v operačním systému MS-DOS je vypsána varovná zpráva, že nemůže být spouštěn pod operačním systémem MS-DOS. Tuto zprávu nastavuje sestavovací program (angl. *linker*) na začátek souboru. Při překladu může být tato zpráva změněna parametrem na libovolný text.

DOSová hlavička je definovaná strukturou `IMAGE_DOS_HEADER`. Důležité jsou zde pouze dvě položky:

- `e_magic` - obsahující signaturu `IMAGE_DOS_SIGNATURE` s hodnotou 0x5A4D. Vyjádřeno v ASCII znacích MZ (Mark Zbikovski - jeden z původních autorů MS-DOSu).

[1] 32-bitová platforma Windows



Obrázek 3.1: diagram Windows PE

- `e_lfanew` - odpovídající odsazení (angl. *offset*) 0x3c, je uložena adresa PE signatury. Tato skutečnost umožňuje systému přímo spouštět programy i přes to, že obsahují DOSovou hlavičku. Adresa (odsazení od začátku programu) signatury PE je nastavena během sestavování programu.

3.1.2 NT hlavičky

Na adrese jejíž hodnota je uložena v položce `e_lfanew` DOSové hlavičky, se nachází struktura `IMAGE_NT_HEADERS` definovaná následovně:

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD Signature   //signatura
  IMAGE_FILE_HEADER FileHeader;
  IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

Signatura

Čtyřbytová hodnota identifikující soubor formátu PE. U platného souboru PE má hodnotu `0x00004550` (definovaná jako `IMAGE_NT_SIGNATURE`). Vyjádříme-li tuto hodnotu v ASCI znacích získáme písmena `P` a `E` následovaná dvěma nulovými byty.

Souborová hlavička

Tato struktura obsahuje základní informace o souboru:

- Typ procesoru - nás zajímají kompatibilní s Intel x86. Tomu odpovídá hodnota 0x14C (`IMAGE_FILE_MACHINE_I386`).
- Počet sekcí - určuje počet záznamů v tabulce sekcí, jenž následuje za hlavičkami NT.
- Časové razítko - datum a čas vytvoření souboru.
- Ukazatel na tabulku symbolů.
- Počet symbolů.
- Velikost hlavičky volitelné hlavičky.
- Charakteristiky - nás zajímají hodnoty `IMAGE_FILE_EXECUTABLE_IMAGE` (EXE soubor) a `IMAGE_FILE_DLL` (DLL knihovna).
- Upozornění, že jsou odstraněny informace pro ladění programu.
- ...

Z hlediska analýzy jsou nejdůležitější charakteristiky souboru a počet sekcí.

Volitelná hlavička

Každý EXE soubor obsahuje tuto hlavičku. Volitelnost hlavičky (viz jméno) vychází z toho, že např. objektové soubory (OBJ) tuto hlavičku neobsahují. Tato hlavička nemá fixní velikost. Její velikost je dána použitou architekturou. lze rozdělit na tři části: standardní, specifické pro Windows a datové adresáře.

Standardní část obsahuje osm položek, které jsou shodné pro PE32 i PE32+. Pole `magic` určuje typ formátu (PE32 - 0x10B/PE32+ - 0x20B). Následují informace o verzi sestavovacího programu, velikosti kódu, velikosti inicializovaných i neinicializovaných dat, vstupním bodu a bázové adrese. Verze PE32 navíc obsahuje bázovou adresu dat. Velikost kódu značí velikost sekce nebo sekcí obsahujících kód. U velikostí dat je to obdobné. Vstupní bod je adresa relativní k bázové adrese, při načtení programu do paměti. Jedná se o startující adresu programu.

Následuje část specifická pro Windows, obsahující infomace o verzích systému a subsystému (GUI - graphical user interface, CUI - console UI), typu subsystému, velikostech (souboru, hlaviček, zásobníku, haldy), zarovnání (sekcí, souboru), kontrolní součet. Na závěr obsahuje počet datových tabulek (`NumberOfRvaAndSizes`), nacházejících se ve zbývající části této hlavičky.

Preferovaná báze programu nahraného do paměti musí být dělitelná 64 kilobyty. Pro DLL je tato hodnota standartně 0x10000000. Pro EXE soubory na Win NT, Win 95 a Win 98 má hodnotu 0x00400000.

Poslední částí této hlavičky jsou datové tabulky. Jedná se o strukturu obsahující relativní virtuální adresu a velikost struktur, reprezentujících příslušné tabulky.

Počet těchto tabulek není pevně daný. Je vyjádřený položkou `NumberOfRvaAndSizes`. Nachází se zde řada zajímavých informací. Nejdůležitější informace jsou umístěny v tabulce adres importovaných funkcí (angl. *Import Address Table* -`IAT`). Hodnoty této tabulky jsou

po zavedení programu do paměti patřičně upraveny podle toho, na jakou relativní virtuální adresu (*RVA*) se zavaděči podařilo načíst jednotlivé moduly (knihovny). Výhodou tohoto uspořádání je, že dochází pouze k úpravě této tabulky a ostatní kód zůstává nezměněn. Tím se práce zavaděče programu podstatně zefektivní. Jedinou nevýhodou tohoto přístupu je využití *IAT* viry pro vyhledání potřebné funkce.

3.1.3 Tabulka sekcí

Každý program je rozdělen do několika sekcí. Každá sekce má určenou adresu ve virtuálním adresovém prostoru, na kterou je v době zavedení umístěna. Tento přístup umožňuje uplatňovat různé politiky v přístupu k datům, případně kódu. Nastavením vlastností můžeme určit ve kterých sekcích lze nalézt spustitelný kód, zamezit přepsání konstantních dat a zajistit další užitečná opatření.

Počet sekcí obsažených v této tabulce obsahuje položka `NumberOfSections` obsažené v souborové hlavičce. Maximálně je možné vytvořit 96 sekcí. Každý řádek tabulky obsahuje jméno, velikost virtuální a skutečnou, *offset* na začátek dat příslušné sekce stejně tak i virtuální adresu. Součástí všech záznamů jsou také ukazatele na relokační tabulky (včetně počtu záznamů v nich obsažených), tabulky s informacemi o řádcích ve zdrojovém souboru a již zmiňované příznaky, určující vlastnosti jednotlivých sekcí.

Jména sekcí nemají žádný vliv na jejich funkčnost. Jsou dána pouze konvencemi. Většinou popisují jaká data sekce obsahuje, ale pojmenování závisí na zvoleném překladači nebo na samotném autorovi programu. Pro analýzu jsou zajímavé oblasti se spustitelným kódem. Jejich nejčastější pojmenování bývá `.text` nebo `.CODE`. Sekce obsahující spustitelný kód mají nastavený některý z těchto příznaků:

- `IMAGE_SCN_CNT_CODE`,
- `IMAGE_SCN_MEM_EXECUTE`.

Je možné sloučit data různých typů do jedné sekce, např. z důvodu optimalizace velikosti programu. Instrukce i data mohou být sloučeny do jedné sekce.

3.1.4 Zavedení programu

Každý program běží ve svém vlastním virtuálním adresovém prostoru. Případná spolupráce programů může probíhat několika způsoby:

- namapování části virtuálního adresového prostoru do virtuálního prostoru jiného programu,
- pomocí dočasných souborů,
- zasíláním zpráv (využití služeb systému),
- pomocí soketů.

Nejprve zavaděč namapuje do virtuálního adresového prostoru jednotlivé sekce a moduly. Pokud možno na jejich preferované adresy. Případně se namapují jinam. Pak je ale nutné upravit tabulku *IAT*, protože volání funkcí se vykonová pomocí této tabulky (vyvolání funkce jejíž adresa je uložena na určitém místě v paměti – položka *IAT*).

Vícevláknové programy většinou obsahují strukturu TLS (z anglického *Thread Local Storage*). Pokud *TLS* obsahuje odkaz na *callback funkci* inicializující data vláken, je

tato funkce vyvolána dříve než je předáno řízení na adresu vstupního bodu programu. Na využití tohoto vstupního bodu poukazuje Peter Szor ve své knize Počítačové viry: analýza a obrana[7].

Připojování knihoven

Pro přehlednost a znovupoužitelnost částí kódu jsou využívány knihovny DLL (angl. Dynamic Link Library). Knihovny lze k programu připojit buď dynamicky nebo staticky. Přestože označení *dynamicky připojované knihovny* tomu neodpovídá.

Statické připojování je zařízeno překladačem během překladu. Deklarace funkcí a proměnných, které knihovna poskytuje, jsou dodány pomocí hlavičkového souboru[2]. Při sestavování programu je využit soubor s příponou LIB[3], která obsahuje všechny informace potřebné k využití knihovny. Například adresy exportovaných funkcí. Kontrolu existence knihovny zajišťuje zavaděč programu. Při dynamickém připojování je kontrola existence knihovny na programátorovi. Využití hlaviček a LIB souborů zůstává téměř stejné. Funkce `LoadLibrary` pak zajistí načtení knihovny. Při sestavování programu s dynamicky linkovanou knihovnou neexistují při překladu adresy funkcí. Získání těchto adres zajišťuje funkce `GetProcAddress`, která tuto hodnotu získá za běhu aplikace. Při dynamickém připojování je knihovna načtena pouze v případě jejího využití. Tím se liší od statického.

Ukázka dynamického linkování:

```
typedef int (*MYPROC)(LPTSTR);

  HINSTANCE h;
  MYPROC ProcAdd;
  h = LoadLibrary(‘‘knihovna.dll’’);  //načtení knihovny
  if (h != NULL) {  //získání adresy funkce
      ProcAdd = (MYPROC) GetProcAddress(h, ‘‘fce’’);
       if (NULL != ProcAdd) { //použití dovezené funkce
        (ProcAdd) (‘‘parametr’’);
      }
FreeLibrary(h);  //uvolnění knihovny z adresy
  }
```

Tento krátký výřez programu načte knihovnu `knihovna.dll` a získá adresu funkce `fce`. Posléze tuto funkci zavolá s příslušnými parametry. Získaná adresa je ukazatelem na funkci. Proto je vhodné ji přetypovat na tzv. signaturu funkce[4]. V ukázce dochází k přetypování uložením hodnoty do proměnné typu ukazatel na funkci, která má parametry a návratovou hodnotu definovaných typů[5] `typedef int (*MYPROC)(LPTSTR);`

Import a export funkcí

Tvůrci programovacího jazyka Microsoft Visual C++ přidali ke standardnímu jazyku C++ několik rozšíření. Některá byla přejata i ostatními překladači, včetně G++ pro Windows. jedním takovým rozšířením je atribut `__declspec` před definicí funkce. Pokud se externí

[2] případně jsou použity dopředné deklarace funkce ve vlastním souboru
[3] na architektuře Windows. Linuxová obdoba má příponu SO
[4] typy parametrů a návratové hodnoty
[5] ukazatel na funkci s parametrem typu LPTSTR a návratovou hodnotou typu int

jména řídí konvencemi jazyka C, musí být použita deklarace `extern ‘‘C’’` v kódu C++. Exportování funkce tedy vypadá takto:

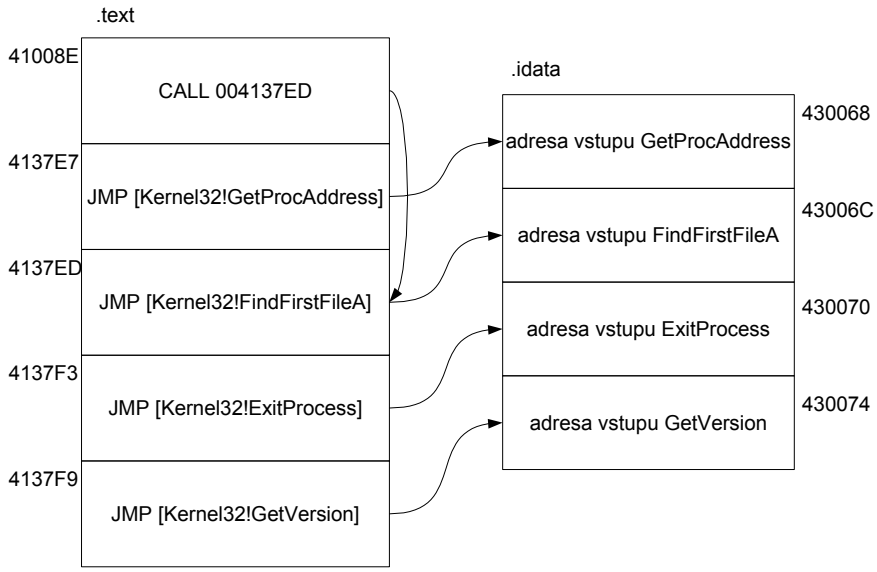
```
extern ‘‘C’’ __declspec(dllexport)double AddNumbers(double a, double b);
```

Importování funkce vypadá následovně:

```
extern ‘‘C’’ __declspec(dllimport)double AddNumbers(double a, double b);
```

Takto explicitně určíme, které funkce budou importovány či exportovány. Při překladu knihovny vznikají soubory DLL a LIB. Soubory LIB jsou využity na připojení knihovny k nějaké aplikaci.

Obrázek 3.2 demonstruje import funkcí. Sekce `.text` obsahuje kód, v sekci `.idata` je uložena tabulka adres importů. Někdy bývá také v `.text` sekci.



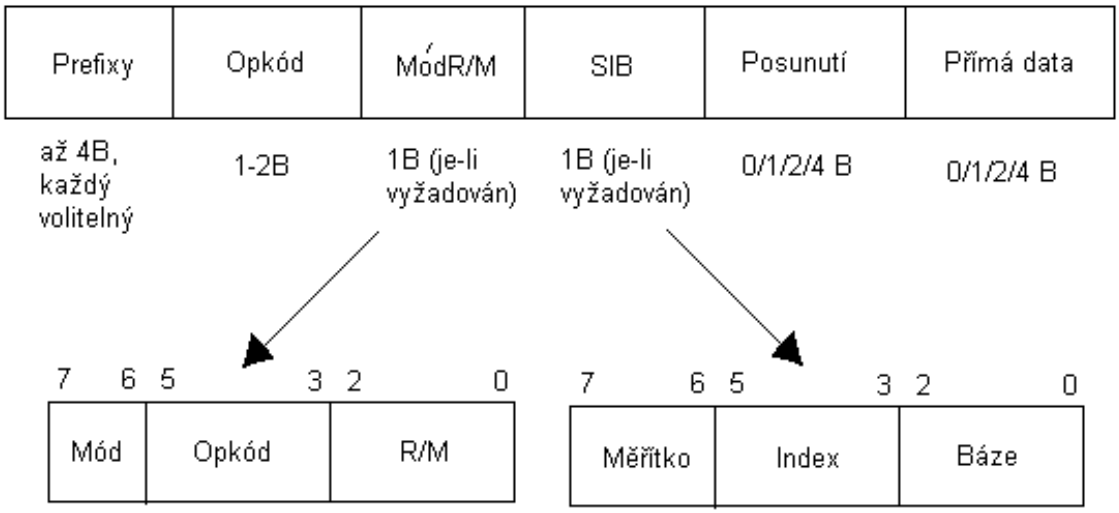
Obrázek 3.2: Import funkce Kernel32!FindFirstFileA

3.2 Jazyk symbolických instrukcí Intel x86

Počítač je schopen komunikovat pouze na nejnižší úrovni (na úrovni bitů a bytů). Příkazy, které má vykonávat, proto musí být ve formě pro něj srozumitelné (jedničky a nuly). Každý příkaz, včetně parametrů, je zakódován na několik bytů. Kvůli srozumitelnosti jsou příkazy a parametry vyjádřeny textovou podobou jazyka symbolických instrukcí. Pro získání tohoto textového vyjádření je nejprve nutné seznámit se s bytovou reprezentací. Základní informace se lze dočíst v článku The Art Of Disassembly (Umění rozebírání)[5] nebo detailněji v manuálu firmy Intel[8].

Každá instrukce lze rozdělit až na šest částí o celkové délce maximálně čtrnáct bytů.

Jak z obrázku 3.3 vyplývá, jedná se o prefixy, operační kód (angl. *opcode* - Operation code), ModR/M byte, SIB byte, posunutí a přímá data.



Obrázek 3.3: Formát instrukce na architektuře Intel

3.2.1 Prefix

Nastavováním prefixů ovlivňujeme chování instrukcí. Prefixy lze rozdělit do několika skupin. Z každé skupiny se může objevit maximálně jeden zástupce[6]. Jejich pořadí není určeno, a proto se mohou v rámci části prefixů vyskytovat v libovolném pořadí.

První skupina mění předdefinovaný segmentový registr. Adresace bude poté probíhat v jiném bloku paměti. Adresový prostor programu bývá zpravidla rozčleněn do několika menších adresových prostorů, nazývaných segmenty. Segmentový registr obsahuje ukazatel na příslušný segment. Na segment kódu ukazuje registr CS (explicitně nastaven prefixem 2Eh). Registr SS (prefix 36h) odkazuje na segment zásobníku. Pro instrukce pracující se zásobníkem (POP, PUSH, ...) se stává standardním segmentem. Zbývající segmentové registry DS(3Eh), ES(26h), FS(64h), GS(65h) jsou určeny pro datové segmenty. Při adresaci dat, pokud není určeno prefixy jinak, je použit segment DS. Výjimku tvoří adresace paměti pomocí registrů EBP, BP, ESP a SP. Pro tyto je základní segment SS. Rozčlenění adresového prostoru na segmenty umožňuje udržovat pořádek v adresování kódu a dat.

Např.

```
kód       textová reprezentace
-------------------------------------
8B00      MOV EAX, DWORD PTR DS:[EAX]
3E:8B00   MOV EAX, DWORD PTR DS:[EAX]
2E:8B00   MOV EAX, DWORD PTR CS:[EAX]
36:8B00   MOV EAX, DWORD PTR SS:[EAX]
26:8B00   MOV EAX, DWORD PTR ES:[EAX]
```

Jak si můžeme všimnout, nestane se nic, pokud prefixem nastavíme segment, jenž je pro danou instrukci implicitní.

Více používaný prefix *VelikostOperandu* (angl. *Operand-Size* - 66h) určuje velikost operandů. Velikost použitého operandu nezávisí pouze na tomto prefixu. Vliv má samozřejmě

[6] maximálně jeden prefix ze skupiny má vliv na chování instrukce

typ operandu, podporovaného použitou instrukcí. Některé typy tímto prefixem nejsou ovlivněny a mají vždy stejnou velikost nezávisle na přítomnosti prefixu *VelikostOperandu*. Pokud je velikost operandu ovlivňována, tak jeho přítomností je zvolena varianta s menší velikostí. Více informací o velikostech operandů bude obsaženo v části o operačních kódech (3.2.2).

Příklad:

```
03C0      ADD EAX, EAX
66:03C0   ADD AX, AX
```

Použití tohoto prefixu zároveň s instrukcí, jenž má přesně danou velikost, nezpůsobí žádnou změnu.

Příklad:

```
00C0      ADD AL, AL
66:00C0   ADD AL, AL
```

Prefix *VelikostAdresy* (angl. *Adress-Size*) určuje použití 16-bitového adresovacího módu na rozdíl od původního 32-bitového.

Příklad:

```
0000      ADD BYTE PTR [EAX], AL    ;32-bitové adresace
67:0000   ADD BYTE PTR [BX+SI], AL  ;16-bitové adresace
```

Změna adresovacího módu se projeví u instrukcí s ukazatelem do paměti jako parametrem. Instrukce, mající jako parametry pouze registry, žádnou změnu nevykážou.

Příklad:

```
00C0      ADD AL, AL
67:00C0   ADD AL, AL
```

Další skupinou jsou prefixy ovlivňující smyčky u řetězcových instrukcí. Použití s jinými instrukcemi nemá žádný efekt. Jedná se o tyto tři prefixy:

- LOCK (F0)
- REPNE/REPNZ (F2)
- REP/REPE/REPZ (F3)

Př.

```
AC      LODS BYTE PTR DS:[ESI]
F2:AC   REPNE LODS BYTE PTR DS:[ESI]
```

Změna chování instrukcí není jedinou úlohou prefixů. U dvoubytových instrukcí prefix slouží k výběru instrukce.

```
0F2900     MOVAPS DQWORD PTR DS:[EAX], XMM0
660F2900   MOVAPD DQWORD PTR DS:[EAX], XMM0
```

Kromě toho, že se změnilo pojmenování instrukce, dochází také ke změně typů parametrů instrukce. Operandy jsou data s pohyblivou řádovou čárkou. V prvním případě jsou s jednoduchou přesností. V případě druhém, s přesností dvojitou.

3.2.2 Operační kód

Za prefixy následuje operační kód (angl. *opcode*), jediná povinná část instrukce. Určuje, která instrukce bude vykonána. Toto pole má velikost jeden až tři byty. Některé SSE nebo SSE2 instrukce jsou dokonce delší. Sada instrukcí je z pohledu programátora pouze dvoudimenzionální tabulka, která obsahuje jména funkcí, *únikové hodnoty* (angl. escape values) a informace o parametrech jednotlivých instrukcí. Bližší informace v kapitole o tabulce operačních kódů. Parametr instrukce je dán buď typem operandu nebo přímo operačním kódem instrukce. Typem operandu je dán např. tento příkaz:

```
00C0    ADD AL,AL
```

První byte říká, o jaký příkaz se jedná a jakého typu jsou parametry. Následujícím bytem je určeno, jestli se jedná o registr nebo o ukazatel do paměti.

Naopak některé instukce přesně určují registr. Ovlivnit lze jenom jeho velikost (AX/EAX). Řada instrukčních kódů mají tu vlastnost, že změnou jediného bitu se vykoná sice stejný příkaz, ale s jinou velikostí operandu či s parametry úplně jiných typů.

Například tato instrukce:

```
40    INC EAX
```

Její bitový format vypadá následovně:

```
0100 0<rrr>
```

Prvních pět bitů značí instrukci INC, další tři bity určují registr. Hodnota `<000>` značí registr EAX.

Těchto vlastností při dekódování instrukce většinou moc nevyužijeme. Snad jen s výjimkou částečné optimalizace dekodéru. Stejně se musíme podívat do tabulky a zjistit jméno instrukce a její parametry. V dekodéru pak zapíšeme, že tyto vybrané instrukce se dekódují přímo z kódu instrukce. Docílíme tak malého snížení počtu typů parametrů.

Skutečnost, že některé parametry jsou zakódovány samotnou hodnotou kódu instrukce, značí výskyt více operačních kódů pro jednu instrukci. Dokonce mohou mít stejné hodnoty parametru.

Příklad:

```
01C0  ADD EAX, EAX
04C0  ADD EAX, EAX
```

Význam mají tyto instrukce stejný, rozdíl je jen ve způsobu zakódování instrukce. V prvním případě je první operand zvláštním případem adresování paměti a druhý operand je přímo registr. U druhé instrukce je tomu obráceně.

Řada instrukcí má stejné typy parametrů. Některé jsou sloučeny do skupin, které mají stejný operační kód, ale jméno instrukce získávají jiným způsobem. Např. stejný operační kód D2 a jiná jména funkcí:

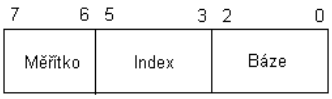
```
D2C0 ROL EAX, CL
D2C8 ROR EAX, CL
D2D0 RCL EAX, CL
```

Kromě dříve jmenovaných zvláštností obsahuje tabulka operačních kódů jednobytových instrukcí únikovou hodnotu (0F), která znamená dvoubytovou instrukci nebo řada instrukcí (hodnota D8 - DF) pro instrukce s pohyblivou řádovou čárkou (FPU).

Další zvláštností operačních kódů jsou různá jména instrukcí pro stejný operační kód. Jedná se o instrukce jejichž vykonáním dosáhneme stejných výsledků. Například využitím příkazu NOP (z angl. No Operation - žádná operace) a XCHG EAX, EAX (výměna hodnot dvou registrů) dosáhneme toho, že stav programu se nezmění. Nic se nestane.

3.2.3 ModR/M

Některé instrukce vyžadují přítomnost bytu ModR/M. Jeho výskyt je dán buď typem parametru nebo příslušností do některé skupiny vyžadující rozšířený operační kód (angl. *opcode extension*). Na základě hodnoty základního i rozšířeného operačního kódu se určuje jméno instrukce. Jedná se o již zmiňované instrukce, mající stejné parametry. Pokud existuje byte ModR/M, nalézá se ihned za operačním kódem. Většina instrukcí jej vyžaduje. S jeho pomocí se určují parametry instrukcí. Byte ModR/M lze brát jako bitové pole, které je rozdělené na několik částí (viz obrázek 3.4).



Obrázek 3.4: ModR/M byte

Mód: zabírá dva nevýznamnější bity. Ve spolupráci s částí R/M získáme 32 různých kombinací, určujících 8 registrů a 24 adresovacích módů.

Pomocí těchto dvou bitů můžeme získat čtyři základní typy adresování.

00 adresa paměti

01 adresa paměti s jednobytovým posunutím

10 adresa paměti s čtyřbytovým posunutím

11 registr

Nyní si ukážeme rozdíly.

```
8B:00           mov eax, dword ptr ds:[eax]
8B:40:00        mov eax, dword ptr ds:[eax+00]
8B:80:00000000  mov eax, dword ptr ds:[eax+00000000]
8B:C0           mov eax, eax
```

R/M: zabírá nejméně významné 3 bity. Určuje registr nebo ve spolupráci s polem `Mód` určuje adresovací mód.

Opkód/Reg: zabírá zbývající 3 bity. Obsahuje volbu registru nebo rozšiřující operační kód.

3.2.4 SIB

K adresování ve tvaru (Báze+Měřítko*Index) je použit byte s označením *SIB* (z angl. *Scale Index Base*). Jak z obrázku 3.5 výplývá, lze jej rozdělit na tři části. Jednu dvoubitovou a dvě tříbitové.

Obrázek 3.5: SIB byte

Měřítko: je umístěno na dvou nejvýznamnějších bitech (6-7) a určuje registr měřítka. Přirovnáme-li jej k poli ve vyšším programovacím jazyce, jedná se o velikost prvku pole. Určuje hodnotu, kterou se bude násobit indexový registr. Tuto hodnotu získáme jako `n`-tou mocninu čísla dvě. Kde `n` je hodnota pole `měřítko`, hledíme-li na něj jako na binární číslo.

```
00: = 2^0 = 1       (add reg,[reg*1])
01: = 2^1 = 2       (add reg,[reg*2])
10: = 2^2 = 4       (add reg,[reg*4])
11: = 2^3 = 8       (add reg,[reg*8])
```

Báze: je umístěna na posledních třech bitech. Určuje bázový registr. V příkladu s polem, by hrál roli ukazatele na začátek pole.

```
SIB byte
00 : 000 : 001       (add reg, [ecx + eax*1])
01 : 001 : 010       (add reg, [edx + ecx*2])
10 : 010 : 011       (add reg, [ebx + edx*4])
11 : 011 : 000       (add reg, [eax + ebx*8])
```

Index: zabírá následující tři bity. Jeho hodnota určuje libovolný indexový registr (v analogii vyššího programovacího jazyka se jedná o index do pole).

```
SIB byte
00 : 000 : ***       (add reg,[eax*1])
01 : 001 : ***       (add reg,[ecx*2])
10 : 010 : ***       (add reg,[edx*4])
11 : 011 : ***       (add reg,[ebx*8])
```

Výjimku tvoří výskyt kódu pro registr ESP (100). V tomto případě jsou registry pro `měřítko` a `index` ignorovány. Využit je pouze registr bázový.

```
032060 ADD EAX, DWORD PTR [EAX]
```

Daný příklad lze chápat takto:

03 ADD

20 ModR/M == 00:100 (SIB):000 (EAX)

60 SIB == 01:100 (Scale*Index ignorovat) :000 (bázový registr == EAX)

3.2.5 Posunutí

Pokud adresový mód[7] obsahuje hodnoty `01` nebo `10`, je posunutí součástí adresy. Nachází se hned za ModR/M bytem. V případě výskytu SIB bytu, se nachází až za ním. Velikost displacementu je jeden, dva nebo čtyři byty.

V tabulce A.1 můžeme vidět rozdíly mezi 32-bitovým a 16-bitovým adresováním. Velikost posunutí závisí na poli `Mód` v bytu ModR/M a na zvoleném adresovém režimu. 8-bitové posunutí je použito v módu `01`, nezávisle na volbě adresování. V režimu `10` se tyto režimy již odlišují. Počet bitů závisí na zvoleném adresovém režimu, tj. 16 nebo 32 bitů.

Posunutí 32 nebo 16 bitů není pouze v uvedených adresových módech (`01` a `10`). Jeden výskyt se nachází také v módu `00`. Určuje jej hodnota pole R/M. V 32-bitovém režimu hodnota `101`. V režimu 16-bitovém hodnota `110`.

Na závěr ještě pár slov k ukládání vícebytového posunutí. Méně významné byty jsou uloženy jako první. Jedná se o tzv. "Little Endian" kódování.

```
          vstup       hodnota
1 byte:   12          12
1 word:   1234        3412
1 dword:  12345678    78563412
```

3.2.6 Přímá data

Jedná se o zadání přímé hodnoty jako parametru instrukce. Beze změn, tak jak je. Například:

```
ADD AL, 10h
```

Stejně jako posunutí má velikost jeden, dva nebo čtyři byty. Hodnota přímých dat nemůže být měněna, protože se jedná o konstantu.

```
1410        adc al, 10
1500000010  adc al, 10000000
0410        add al, 10
0500000010  add eax, 10000000
```

3.2.7 Tabulka operačních kódů

Základní funkce a využití jsme si nastínili v kapitole 3.2.2. Nyní si probereme detailněji její použití. Jak již bylo zmíněno, jedná se o dvoudimenzionální tabulku. Horní čtyři bity uvažované jako binární číslo jsou indexem řádku tabulky. Dolní čtyři bity indexují sloupce. Ukázka opcode tabulky je v tabulce 3.1

Obsahem každého políčka je jméno funkce a typy parametrů včetně jejich velikostí. Každá instrukce má nula až tři parametry. Získání hodnoty parametru nezáleží na pozici operandu. Parametr se tedy dekóduje stejným způsobem jak na prvním, druhém tak i na třetím místě.

[7] Mód pole bytu ModR/M

[3] použit byte SIB

Tabulka 3.1: Výřez 1 bytové tabulky operačních kódů

	0	1	2	3	4	5	6	7
0	ADD						PUSH	POP
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	ES	ES
1	ADC						PUSH	POP
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SS	SS
2	AND						SEG=ES	DAA
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	(prefix)	
3	XOR						SEG=SS	AAA
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
...								
D	Shift Grp 2				AAM	AAD		XLAT
	Eb, 1	Ev, 1	Eb, CL	Ev, CL	Ib	Ib		
...								

První znak vyjadřující typ parametru je velké písmeno, které určuje způsob adresování parametru - jestli se jedná o adresu v paměti, registr a případně jeho druh. Druhý znak udává velikost parametru. V případě registrů určuje použití např. AL, AX, EAX,... Při adresování paměti určuje velikost paměti, na kterou adresa odkazuje. Zapisováno např. DWORD PTR [...].

Metody adresování lze rozdělit podle různých kritérií do různých skupin. Podle použité technologie či použití:

- obecného použití,
- MMX technologie,
- XMM technologie,
- Ladicí (angl. *Debug*) registry,
- Kontrolní (angl. *Control*) registry,
- a další.

Podle způsobu zakódování:

- reg pole ModR/M bytu,
- R/M pole ModR/M bytu,
- přímá adresa,
- přímá hodnota,
- přímo jméno registru.

Registry jsou zakódovány buď v poli `reg` ModR/M bytu. Případně v `R/M` poli pokud má `mod` pole hodnotu `11`. Kód registru je 3 bitové binární číslo. V podstatě se jedná o index do tabulky A.2.

Nyní jsme se již seznámili se všemi potřebnými postupy. Můžeme si tedy vyzkoušet něco dekódovat. Mějme jednodušší příklad. Kód instrukce je vyjádřen hexadecimálním číslem `26004710`.

26 odpovídá nastavení segmentu na ES

00 je operační kód. V tabulce 3.1 zjistíme, že se jedná o instrukci ADD s parametry typu Eb, Gb. Typ E značí obecný registr nebo paměť s využitím indexových a bázových registrů, případně i posunutí. Typ G určuje využití pouze obecných registrů. Malé b značí velikost operandu jeden byte. Oba operandy vyžadují přítomnost bytu ModR/M.

47 ModR/M bitová reprezentace - (01 : 000 : 111)b
podíváme se do tabulky A.1, kde zjistíme, že mod == (01)b značí 8bitový displacement
reg == (000)b značí registr EAX
R/M == (111)b značí registr EDI

10 8 bitové posunutí

Výsledkem je: `ADD BYTE PTR ES:[EDI+10],AL`

Další příklad `6601845000000010`, demonstruje využití bytu SIB.

66 prefix VelikostOperandu.

01 instrukce ADD s parametry Ev, Gv. Velikost operandu označená písmenem `v`, určuje 16 nebo 32 bitový operandy. V závislosti na přítomnosti prefixu VelikostOperandu. V našem případě tedy 16 bitové hodnoty.

84 ModR/M == (10 : 000 : 100)b
mod == (10)b značí 32 bitový displacement
reg == (000)b značí registr AX (16 bitový registr)
R/M == (100)b značí přítomnost SIB bytu.

50 SIB == (01 : 010 : 000)b
Měřítko == (01)b značí hodnotu 2
Index == (010)b značí registr EDX[8]
Báze == (000)b značí registr EAX

00000010 32 bitový posunutí == 10000000

Výsledkem je instrukce `ADD WORD PTR DS:[EAX+EDX*2+10000000],AX`. Protože jsme explicitně neurčili volbu segmentového registru, byl pro tuto instrukci implicitně zvolen segment DS.

[8]prefix VelikostOperandu nemá vliv na šířku registru. Na výběr registru má vliv ještě prefix VelikostAdresy, který určí volbu 16 bitových

Poslední ukázka (`670100`) předvede využití 16 bitového adresování.

67 prefix VelikostAdresy.

01 instrukce ADD s již známými parametry Ev, Gv. Oba operandy jsou 32 bitové.

00 ModR/M == (00 : 000 : 000)b
mod == (00)b mod bez displacementu. reg == (000)b registr EAX R/M == (000)b určuje hodnotu `BX + SI`, protože máme nastavené 16 bitové adresování.

Výsledkem je `ADD DWORD PTR DS:[BX+SI], EAX`

3.3 Graf toku řízení

Základní informace o grafech toku řízení jsem získal z přednášek prof. Alexandra Meduny [4]. Cenným zdrojem informací z oblasti analýzy grafů je disertační práce Dr. Cristiny Cifuentes [3].

Graf toku řízení je diagram, který znázorňuje možné posloupnosti předávání řízení jednotlivým blokům kódu.

Využití grafů toků řízení je především v oblasti překladačů a analýzy programů. Mohou vést k optimalizaci generovaného kódu odstraněním nedostupných bloků, detekci potenciálně nebezpečných konstrukcí, ...

Grafy toků řízení lze rozlišit jako strukturované a nestrukturované. Strukturovaný graf je tvořen bloky, jenž mají jeden vstupní bod a jeden či více výstupních bodů. Nestrukturovaný graf obsahuje bloky, jenž mají jeden vstupní bod a jeden výstupní bod. Ve své práci se zabývám nestrukturovanými grafy.

Stavebními kameny takového grafu jsou *základní bloky*. Jedná se o posloupnosti příkazů. Jediným vstupním bodem tohoto bloku je jeho začátek. Jediným výstupním bodem je konec základního bloku. Díky těmto vlastnostem není možný skok doprostřed bloku ani vnořování dalších bloků. Na bloky lze tudíž pohlížet jako na atomické operace.

3.3.1 Vedoucí příkazy

Prvním krokem získání základních bloků je získání vedoucích příkazů. Proto si je nejprve definujeme:

- první příkaz programu je vedoucí,
- každý příkaz, který je návěštím pro příkazy skoku, je vedoucím příkazem,
- každý příkaz, který následuje za podmíněným příkazem skoku, je vedoucím.

V běžně známé definici je vedoucím příkazem také příkaz, vyskytující se za instrukcí nepodmíněného skoku. Pro případ binárního souboru toto striktně neplatí. Příkazy jazyka symbolických instrukcí jsou v souboru reprezentovány jako posloupnost bytů. Je možné pomocí instrukce skoku předat řízení na libovolné místo programu. Díky těmto možnostem a kvůli zarovnávání, které zajistí překladač při generování binárního souboru, vznikají v kódu mezery. Některé překladače tyto mezery vyplňují bytem s hodnotou CC. Tato hodnota odpovídá instrukci INT (přerušení). Některé překladače využívají hodnoty 90 (instrukce NOP). Důsledkem tohoto je správný výpis disasembleru. Jednobytová výplň zajistí, že při lineárním procházení binárního kódu a jeho dekódováním nedojde k chybnému

posunu při dekódování instrukce, která odpovídá náhodné výplni. Náhodná výplň může způsobit přečtení několika bytů, které jsou již součástí platného kódu. Řada disasemblerů této vlastnosti jednobytové výplně mezer využívá.

Pospojováním dostatečného množství těchto mezer může získat útočník místo pro vložení vlastního kódu. Vložením některých hodnot na začátek mezery způsobí, že navenek se oblast mezery tváří jako určitý kód, ale vlivem posunu obsahuje ve skutečnosti kód úplně jiný.

3.3.2 Základní bloky

Definice základního bloku:

- Základní blok začíná vedoucím příkazem.
- Základní blok končí příkazem skoku:
  - podmíněné i nepodmíněné příkazy skoku (JMP, JCXZ, ...),
  - volání funkcí instrukcí CALL,
  - návrat z funkce instrukcí RET.
- Základní blok končí příkazem, který předchází příkazu vedoucímu.

3.3.3 Graf toku řízení

Pro získání grafu si nejprve zajistíme seznam vedoucích příkazů. S jejich pomocí rozdělíme kód na základní bloky. Graf vznikne propojením získaných základních bloků. Toho docílíme přidáním orientovaných hran. Pokud je blok ukončen instrukcí skoku, směřuje hrana na blok, jehož vedoucí příkaz je cílem skoku. Některé bloky nejsou ukončeny instrukcí skoku, protože těsně následující blok vznikl jako cíl nějakého skoku. Orientovaná hrana se tedy přidá i mezi tyto bloky.

3.3.4 Vizuální vyjádření grafu toku řízení

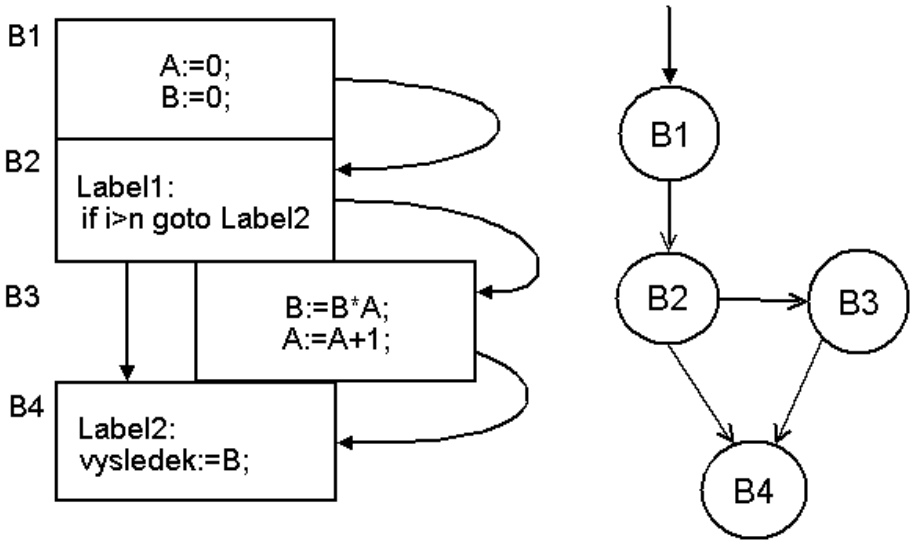
Graf toku řízení lze vyjádřit jako matematickou strukturu, nazývanou orientovaný graf. Jeho vizuální vyjádření tomu odpovídá. Vrcholy grafu značí základní bloky. Orientované hrany vyjadřují možné posloupnosti vykonávání bloků kódu.

Šipkou je též označen počátek grafu. Obrázek 3.3.4 ilustruje převod kódu programu na graf toku řízení.

3.3.5 Možnosti využití

Analýzou zdrojových kódů, případně programů, jsme do jisté míry schopni získat graf toku řízení. V tomto grafu jsme schopni detekovat struktury, které odpovídají základním operacím jazyka C, případně další analýzou detekujeme potenciálně nebezpečný kód.

Pomocí existujího grafu toku řízení jsme schopni získat přesnější výpis strojových instrukcí. Začneme soubor zpracovávat od vstupního bodu. Při větvení grafu si zapamatujeme reference začátků ostatních větví. Po dokončení zpracovávané větve pokračujeme s některou uloženou referencí. Můžeme se tak vyhnout chybnému dekódování a zvýšit šance na odhalení samopřepisovatelných kódů a jiných potenciálně nebezpečných konstrukcí.



Obrázek 3.6: Převod kódu na graf toku řízení

3.3.6 Získání grafu toku řízení

Při použití statické analýzy je získání plnohodnotného grafu toku řízení z binárních souborů EXE nemožné. Brání tomu vypočítané skoky, tzv. pozdní vazba. Bez emulace programu nejsme schopni určit cíl skoku. Následkem je, že graf může pokračovat prakticky kdekoliv v programu.

Existují dva postupy získání grafu toku řízení ze seznamu instrukcí. Metoda zdola-nahoru slučuje instrukce do základních bloků a metoda shora-dolů, která blok instrukcí rozčleňuje na základní bloky.

Kapitola 4

Analýza

4.1 Analýza tvorby disasembleru

Rozpoznání instrukcí ve spustitelném souboru EXE je nutným krokem pro jeho další analýzu. Mezi nejznámnější analyzátory kódu patří `IDA`[1] a `OllyDbg`[2]. Vytvoření vlastního analyzátoru umožňuje využití informací, které jsou získány během získávání textové podoby instrukcí.

Algoritmus 1: Nalezení sekce kódu

Vstup: Vstupní bod programu
Výstup: Ukazatel na začátek kódové sekce v souboru

```
1 for i=0..počet sekcí-1 do
2 |   if Vstupní bod ∈ virtuálnímu adresovému prostoru sekce[i] then
3 |   |   return ukazatel na začátek sekce[i] v souboru
4 |   end
5 end
```

4.1.1 Nalezení místa dekódování

Prvním úkolem disasembleru je nalezení místa v souboru kde jsou uloženy instrukce. Informace k tomu potřebné se nacházejí v hlavičkách souboru. Detailně byly probrány v kapitole 3.1.

Při dekódování můžema postupovat dvěma způsoby:

- Nalezneme první instrukci pomocí hodnoty adresy vstupního bodu. Další dekódování pak ovlivňuje tok řízení zkoumaného programu.
- Nalezneme začátek sekce s kódem. Od začátku do konce této sekce sekvenčně procházíme a dekódujeme. Tento přístup je používanější.

Nezávisle na zvoleném postupu je nutné určit místo v souboru, kde začíná sekce kódu. Za předpokladu, že jsou dostupné potřebné informace, lze začátek sekce kódu nalézt algoritmem 1.

Algoritmus 2: dekódování instrukcí

Vstup: posloupnost bytů
Výstup: dekódované instrukce

```
1  while není dekódováno vše do
2      repeat
3          tmp=přečti další byte;
4          if tmp je prefix then
5              //nastavení příznaku výskytu prefixu;
6              zaznamenat výskyt tohoto prefixu
7          end
8      until tmp je prefix ;
9      switch tabulka_1B[tmp] do
10         case normální
11             dekóduj parametry
12         end
13         case 2B ESC
14             tmp=přečti další byte;
15             switch tabulka_2B[prefix][tmp] do
16                 case normální
17                     dekóduj parametry
18                 end
19                 case GRP
20                     dekóduj parametry a jméno pomocí rozšiřující tabulky
21                 end
22                 otherwise
23                     nedělej nic
24                 end
25             end
26         end
27         case FPU ESC
28             dekóduj FPU instrukce pomocí tabulek pro FPU
29         end
30         case GRP
31             dekóduj parametry a jméno pomocí rozširující tabulky
32         end
33         otherwise
34             nedělej nic
35         end
36     end
37 end
```

4.1.2 Postup dekódování

Postup dekódování lze popsat algoritmem 2. Nejprve se přečtou a zaznamenají všechny prefixy. Další byte slouží jako index do tabulky, ve které se nacházejí informace o jednobytových instrukcích. Některá pole mají zvláštní význam, slouží k přepnutí na jiný způsob dekódování (např. dekódování FPU instrukcí).

Při dekódování FPU ovlivňuje volbu tabulky hodnota bytu, který přepnul na FPU instrukce. Všechny FPU instukce vyžadují přítomnost bytu ModR/M. Nejvyšší dva bity ovlivňují použití paměti nebo registrů. Jinak se použití tabulek pro FPU nijak neliší od normální jednobytové tabulky.

4.1.3 Parametry instrukcí

Základní informace o parametrech byly probrány v kapitole o tabulce operačních kódů (3.2.7). Nyní je probereme detailněji s ohledem na stavbu disasembleru. Vyjádření typu parametru v tabulce operačních kódu lze rozdělit do dvou skupin.

První skupina zadává parametr přímo hodnotou, tj. číslo nebo jméno registru (např. 1, 3, EAX, AX, eAX, rAX). Hodnoty eAX a rAX jsou ovlivňovány přítomností prefixu VelikostOperandu. Při jeho výskyty mají význam AX, jinak EAX.

Druhá skupina je určena dvojicí písmen. První písmeno je velké a určuje způsob adresování. Typ adresování určuje přítomnost bytu ModR/M. Druhé písmeno je malé a určuje velikost operandu. I v této skupině je velikost operandu ovlivňena přítomností prefixu VelikostOperandu. Problém velikosti operandu je možné převést na výběr z tabulky se dvěma řádky. V jednom řádku bez prefixu VelikostOperandu a v druhém řádku s ním.

Adresovací metody lze rozdělit do několik skupin:

- přímá hodnota - přímá adresa, relativní posunutí v paměti (hodnota přidaná k registru ukazatele instrukcí),
- registr,
- pamět,
- registr nebo paměť (závisí na hodnotě pole Mód bytu ModR/M).

Přímá hodnota značí, že za bytem ModR/M (v případě jeho výskytu bytu SIB, tak i za ním) se nachází přímá hodnota dat, jejichž velikost je dána velikostí operandu.

K určení jména registru může být použita hodnota polí `reg` nebo `registr/paměť` bytu ModR/M. Výběr registru lze reprezentovat dvoudimenzionální tabulkou. První dimenze značí typ registru (kontrolní, ladicí, MMX, XMM a obecného použití). Druhá dimenze určuje přímo konkrétní registr.

Dekódování paměti je nezávislé na použité technologii (MMX, XMM, ...). Probíhá stejným způsobem jak pro typ "paměť" tak pro typ "registr / paměť".

Volba registru je také speciálním případem typu "registr / paměť" (hodnota (11)b je obsažena v poli Mód bytu ModR/M). I v tomto případě se registry dekódují stejným způsobem jako u samotného typu registr.

Určování hodnot těchto typů lze tedy rozdělit na problém získání registru a problém získání adresy paměti. Tím se tvorba disasembleru zjednodušší.

Dekódování paměti a registrů bylo detailněji probráno v kapitole 3.2.

```
00401025: BE01000000            MOV  ESI,00000001
0040102A: C70424080000          MOV  DWORD PTR [ESP],00000008
          00
00401031: 31C0                  XOR  EAX,EAX
00401033: 89442404              MOV  DWORD PTR [ESP+4],EAX
00401037: E8A4070000           CALL  004017E0
0040103C: 83F801                CMP  EAX,01
0040103F: 746C                   JE  004010AD
```

Obrázek 4.1: Ukázka textového výstupu disasembleru

4.1.4 Výstup programu

Dále bylo nutné vyřešit otázku formátu výstupu disasembleru. Jaké informace v něm budou obsaženy? Zde jsem se inspiroval formátem textového výstupu konzolového disasembleru `dumpbin`, který kromě výpisu dekódovaných instrukcí dokáže vypisovat i obsah hlaviček souborů v PE formátu (viz obrázek 4.1).

První sloupec obsahuje virtuální adresu instrukce. Druhý sloupec obsahuje kód instrukce. Pokud je kód příliš dlouhý, je rozdělen a výpsán na dalším řádku. Ve třetím sloupci je jméno funkce. Parametry se nacházejí ve sloupci posledním. Všechna čísla jsou vyjádřena v šestnáctkové soustavě.

Při vytvoření instance třídy dekodéru je zadán výstupní stream, do kterého jsou data vypisována.

4.2 Analýza grafu

V jazyce symbolických instrukcí je zápis základních programátorských konstrukcí (cykly a větvení) poněkud rozsáhlejší a tudíž i méně přehledný. Jednoduchý podmíněný příkaz může mít vyhodnocení a jednotlivé větve rozmístěném po celém souboru. Cílem analýzy grafu je detekovat tyto základní konstrukce vyššího programovacího jazyka a usnadnit pochopení zkoumaného kódu a jeho následnou analýzu.

4.2.1 Získání grafu

Při procesu získávání grafu toku řízení potřebujeme znát hodnoty cílů skoku. Prvním krokem při sestavení grafu toku řízení je rozdělení dekódovaných instrukcí na základní bloky. Je výhodnější část této práce vykonat již při běhu disasembleru. Zamezí se tím analýze výstupních dat. Když disasembler při svém běhu dojde k instrukci skoku, zaznamená si výskyt a typ skoku (podmíněný/nepodmíněný). Současně si uložíme cíle skoků.

Po dokončení dekódování programu začíná algoritmus, který za pomoci uložených informací získá základní bloky. Dochází k dělení zkoumaného prostoru na základě infromací o některých začátcích bloků a některých konců. Po rozdělení zůstávají disjunktní intervaly (základní bloky).

4.2.2 Postup analýzy

Nyní začíná samotná analýza grafu. Základní bloky jsou reprezentovány vrcholy grafu. Nalezení struktur vyšších programovacích jazyků odpovídá hledání podgrafu. Hledají se struktury, které mají jisté vlastnosti, a nahrazují se jednodušší strukturou. Většinou jediným vrcholem grafu. Tento postup se nazývá redukce grafu.

Během redukce grafu hledáme i takové struktury, které přímo nejsou hledaným větvením nebo cyklem. Využijí se pro zjednodušení grafu. Malé jednoduché části se lépe rozpoznávají.

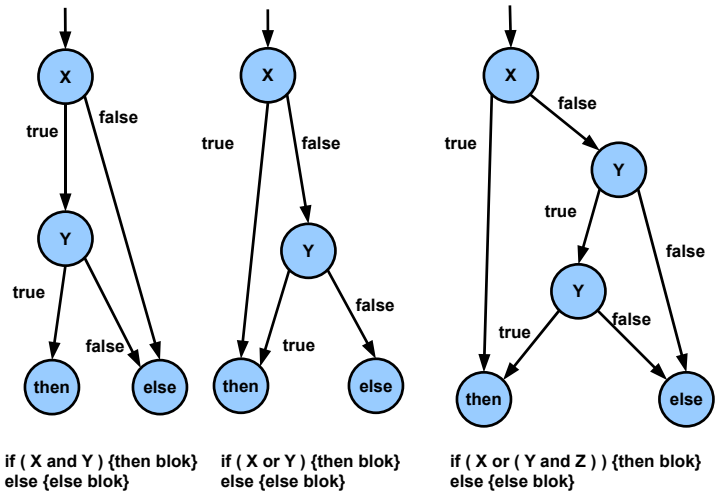
Redukce grafu probíhá tak dlouho, dokud jsou v grafu rozpoznávány vzory.

4.2.3 Vyhodnocování podmínek

Podmínky a jejich vyhodnocování jsou základem každého strukturovaného jazyka. Ať se jedná o podmíněné větvení nebo příkazy cyklu, vždy je nutné vyhodnocení nějakých podmínek. Buď k určení větve příkazu `if-then-else` nebo k ukončení cyklu `for`, `while` nebo `do-while`.

Jednodušší podmínky se nejprve vyhodnotí. Na základě jejich hodnoty je určen cíl podmíněného skoku. Tento přístup začíná být neefektivní za použití složitějších podmínek. Některé části vyhodnocované podmínky není nutné počítat, protože výsledek celého logického výrazu je již známý. Proto se využívá tzv. zkrácené vyhodnocování booleových podmínek. Má-li např. při logickém součinu `x AND y` jedna složka hodnotu false, celý výraz má hodnotu false a druhá složka se již nevyhodnocuje. Tento přístup vede k optimalizaci rychlosti programu. Použití zkráceného vyhodnocení demonstruje obrázek 4.2.

Také příkazy větvení a smyček jsou realizovány pomocí podmíněných a nepodmíněných skoků. Instrukce podmíněného skoku se rozhodují na základě nastavených bitů příznakového registru. Způsobů jak tedy v jazyce symbolických instrukcí zapsat podmínku existuje řada. Velmi často jsou využívány instrukce porovnání (`CMP` a `TEST`), logické (`AND, OR, XOR`) nebo dokonce aritmetické `ADD, SUB, DEC, INC`. Všechny tyto instrukce nastavují příznaky (nulová hodnota, záporná hodnota, ...).



Obrázek 4.2: Vícenásobné podmínky

Na základě těchto příznaků se rozhodují instrukce podmíněného skoku:

- `JE` - skok při nastaveném příznaku nuly,
- `JNE` - skok při nenastaveném příznaku nuly,

Zvláštním případem podmíněného větvení jsou varianty instrukce `LOOP`. Tato instrukce dekrementuje obsah registru čítače (CX/ECX), aniž by došlo ke změně nastavení hodnot příznaků. Na základě obsahu registru čítače, případně také hodnoty příznaku *ZF* (z angl. *Zero Flag* - příznak nuly), vykoná skok na cílovou adresu.

Varianty instrukce `LOOP` zohledňující hodnotu příznaku *ZF* jsou tedy schopny vykonávat cykly se složitějšími ukončovacími podmínkami.

4.2.4 Podmíněné větvení

V jazyce C jsou příkazy větvení `switch` a `if-then-else`, přičemž část `else` je volitelná. Takto může vypadat `if-then` a `if-then-else` zapsané v asembleru s jednoduchou podmínkou.

```
    ;if(){}
      CMP EAX,1
      JNZ L1
      <kód větve then>
L1:
```

```
    ;if(){}else{}
      CMP EAX,1
      JNZ L1
      <kód větve then>
      JMP L2
L1:
      <kód větve else>
L2:
```

Příkaz `switch` lze reprezentovat různě. Buď pomocí struktury `if-then-else`:

```
switch(i){          if (i==1) {              CMP EAX,1
  case 1:             <případ1>                    JNE L2
    <případ1>       } else {                 <případ1>
    break;            if (i==2) {            JMP L4
  case 2:               <případ1>             L2: CMP EAX,2
    <případ2>         } else {                 JNE L3
    break;              if (i==3) {            <případ2>
  case 3:                 <případ1>                 JMP L4
    <případ3>           }                  L3: CMP EAX,3
    break;            }                        JNE L4
}                   }                          <případ3>
                                             L4:
```

nebo pomocí cyklu s využitím tabulky skoků. Tento přístup je vhodný především při větším množství větví `case` s hodnotami, jenž po sobě následují. Např. hodnoty 0-30, kdy získání nové testované hodnoty získáme inkrementací či dekrementací. Při dosti odlišných hodnotách lze testované hodnoty uložit také do tabulky.

Pokud detekujeme strukturu `if-then`, hledáme dva vrcholy grafu. Oba mají jednou výstupní hranu, směřující na shodný třetí vrchol. První vrchol má navíc hranu, která směřuje na druhý vrchol (jediná vstupní hrana vrcholu).

Tyto dva vrcholy nahradíme vrcholem jediným. Výstupní hrana směřuje na třetí vrchol. Množina vstupních hran je rovna množině vstupních hran prvního vrcholu.

Detekování `if-then-else` je také snadné. Hledám vrchol grafu, který má dvě výstupní hrany. Tyto hrany jsou jedinými vstupními hranami cílových vrcholů. Cílové vrcholy mají jedinou výstupní hranu směřující na společný čtvrtý vrchol. Všechny tři vrcholy jsou nahrazeny jedním vrcholem. Výstupní hrana směřuje na čtvrtý vrchol. Množina vstupních hran je rovna množině vstupních hran prvního vrcholu.

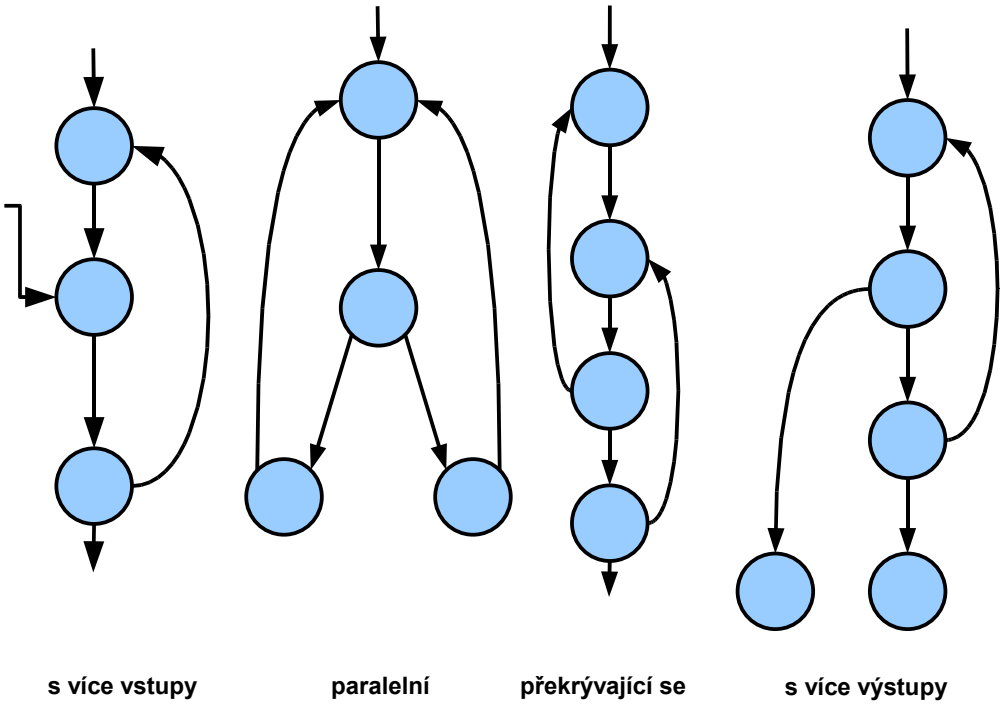
4.2.5 Smyčky

Jednou ze základních struktur je smyčka. Jedná se o graf, který má jednu či více zpětných hran a jeden či více výstupních bodů. Obrázek 4.3 demonstruje základní typy smyček:

- s více vstupními body,
- paralelní smyčky (více zpětných hran do stejného cíle),
- překrývající se smyčky,
- s více výstupními body.

Dále se však budeme zabývat strukturami grafů, které odpovídají smyčkám v jazyce *C*. V jazyce C existují tři druhy cyklů:

- for cyklus,
- while cyklus,
- do-while cyklus.



Obrázek 4.3: Ukázka cyklů

Ve skutečnosti lze z programu rozeznat pouze cykly `while` a `do-while`. Cyklus `for` je v jazyce symbolických instrukcí totožný s cyklem while (srovnejte - ukončující podmínka, počáteční inicializace a přechod do dalšího cyklu).

Nejjednodušeji lze rozeznat smyčku `do-while`. Jedná se o blok zakončený podmíněným skokem. Cíl skoku směřuje na začátek toho samého bloku. Tento blok je nahrazen novým blokem. Množina vstupních bodů je proti původnímu bloku zmenšena o hranu začínající a končící ve stejném vrcholu. Množina výstupních bodů obsahuje pouze jednu hranu. Hranu směřující na blok následující za podmíněným skokem (nesplněna podmínka skoku) nastavíme jako výstupní hranu nového bloku.

Dalším cyklem je `while-do`. Často je implementován stejně jako do-while s počátečním skokem na vyhodnocování podmínky zastevení cyklu. V grafu jej detekujeme jako dva vrcholy oběma směry propojenými. První z nich má pouze jednu vstupní a jednu výstupní hranu (tj. hrany jej spojují s druhým vrcholem). Tyto dva vrcholy jsou nahrazeny vrcholem jediným. Množinu vstupních hran použijeme z druhého vrcholu. Odstraníme z ní hranu vedoucí z vrcholu prvního. Výstupní hranou nového vrcholu je druhá hrana druhého vrcholu (tj. ta co není součástí smyčky).

Kromě těchto základních lze detekovat zvláštní případy cyklu:

- nekonečná smyčka

```
for(;;)
{
  <kód>
}
```

  Lze ji detekovat jako vrchol, jehož jediná výstupní hrana směřuje na ten samý vrchol. Nahradíme jej novým vrcholem, který má vstupní hrany stejné jako vrchol původní (vyjma hrany smyčky). Výstupní hrany nemá žádné.

- smyčka break

```
while(1)
{
  <kód>
  if(<podmínka>)
  {
    <kód>
    break;
  }
}
```

  Příkaz `break` abnormálně ukončí cyklus. Detekovaný podgraf obsahuje dva vrcholy se dvěmi výstupními hranami. Oba jednou hranou směřují ke stejnému třetímu vrcholu. Hrana z prvního vrcholu je jedinou vstupní hranou vrcholu druhého. První i druhý vrchol je nahrazen vrcholem novým. Množina vstupních hran je rovna množině vstupních hran prvního vrcholu. Množina výstupních hran je rovna množině výstupních hran druhé vrcholu.

  Pomocí této struktury lze redukovat některé složitější větvení. Detekci příkazu `break` docílíme přidáním podmínky na první vrchol. Ten musí být začátkem cyklu, ve kterém se tato struktura nachází.

- smyčka s příkazem `continue`

```
for(i=0;i<30;i++)
{
  <kód>
  if(i%5)
  {
   <kód>
   continue;
  }
  <kód>
}
```

Příkaz `continue` ukončí probíhající cyklus smyčky a skočí na začátek nového cyklu. Hledaný podgraf obsahuje dva vrcholy, vzájemně propojené hranami. První vrchol je počátkem cyklu a svoji druhou hranou směřuje za cyklus. Druhý vrchol má jedinou vstupní hranu a směřuje na vrchol obsažený ve zkoumané smyčce. Tento podgraf je nahrazen jediným vrcholem. Množin vstupních hran obsahuje množinu vstupních hran druhého vrcholu (bez hrany obsažené ve smyčce).

V jazyce symbolických instrukcí pak tyto základní cykly mohou vypadat například takto:

```
;while
      L2:
         <vyhodnocení podmínek>
         JA L1
         <tělo cyklu>
         JMP L2
      L1:
```

```
;do-while
L1:
  <tělo cyklu>
  JL L1
```

Kapitola 5

Implementace

5.1 Implementace disasembleru

Implementačním jazykem celého projektu je C++. Téměř všechny informace, které jsou nutné ke správnému dekódování instrukce jsou obsaženy ve formě tabulek. Tento přístup jsem zachoval i při implementaci. V některých případech, kdy se jednalo o tzv. řídké pole jsem učinil výjimku. Ale i zde je na zvážení, jestli by tabulkový přístup nebyl efektivnější. Využitím tabulek, implementovaných jako jednorozměrné a dvourozměrné pole, jsem omezil použití pomalejších příkazů větvení `switch`.

5.1.1 Načtení do paměti

Existují dva možné přístupy získávání dat ze souboru. Otevření souboru a jeho sekvenční procházení[1] nebo načtení souboru do paměti. Druhá varianta je vzhledem k načítaným strukturám efektivnější.

Nejprve je nutné otevřít soubor pro čtení. Toto zajišťuje funkce `CreateFile` s parametrem jméno souboru (filename). Pomocí funkce `CreateFileMapping` a funkce `MapViewOfFile` zajistíme načtení souboru do paměti např. takto:

```
HANDLE hF;
hF=CreateFile((LPCTSTR)filename,GENERIC_READ,FILE_SHARE_READ,
              NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0);
hFM=CreateFileMapping(hF,NULL,PAGE_READONLY,0,0,NULL);
PBYTE baze=(PBYTE)MapViewOfFile(hFileMapping,FILE_MAP_READ,0,0,0);
```

Vhodné doplnit testování návratových hodnot jednotlivých funkcí. Obsah souboru se nyní nachází v paměťovém prostoru programu. Proměnná `baze` nyní obsahuje ukazatel do paměti, kde je nyní dostupný obsah souboru. V případě většího souborů se nemusí v paměti nacházet celá jeho kopie. V případě požadavku na část souboru, která se nenachází v paměti, je režie spojená s jeho načtením vykonána automaticky. Při ukončení práce se souborem je korektní uvolnění jeho paměťové reprezentace zajištěno funkcemi `CloseHandle` a `UnmapViewOfFile`.

Po namapování do paměti je se souborem pracováno jako s pamětí. Např. požadujeme hodnotu, která má určitou vzdálenost od počátku souboru. Sečtením této vzdálenosti a *báze souboru*[2] získáme hodnotu ukazatele na požadované místo v paměti. Přístup k hlavičkám

[1] případně větší skok v souboru

[2] adresa souboru v paměti

se stává jednoduchým výpočtem a přetypováním na ukazatel na strukturu, která tuto hlavičku reprezentuje.

```
PIMAGE_DOS_HEADER g_pDOS=0;
PIMAGE_NT_HEADERS g_pNT=0;
PIMAGE_FILE_HEADER g_pFH=0;
PIMAGE_SECTION_HEADER sections=0;
/*DOSova hlavicka*/
g_pDOS=(PIMAGE_DOS_HEADER)g_pMappedFileBase;
/* NT hlavicka */
g_pNT=(PIMAGE_NT_HEADERS)((PBYTE)g_pMappedFileBase+g_pDOS->e_lfanew);
/* File Header*/
g_pFH=(PIMAGE_FILE_HEADER)&(g_pNT->FileHeader);
/* ukazatel na prvni sekci */
sections=(PIMAGE_SECTION_HEADER)((PBYTE)g_pNT+sizeof(IMAGE_NT_HEADERS));
```

Tímto jednoduchým způsobem jsou zpřístupněny základní struktury EXE souboru.

5.1.2 Reprezentace tabulky operačních kódů

Důležitými částmi disasembleru jsou tabulky operačních kódů. Musí obsahovat jména funkcí a typy parametrů. Dále musí obsahovat dodatečné informace jako je příslušnost do skupiny instrukcí, které se dekódují pomocí tabulek rozšířených operačních kódů. Nejprve jsem definoval typ, který reprezentuje typ operandu. Ten je složen z typu adresy a velikosti operandu.[3] Přímé hodnoty operandů (např. EAX nebo 1) jsou zadány jako typ adresy.

```
typedef struct structType {
  BYTE am; //adresovaci metoda
  BYTE ot; //typ operandu (velikosti)
  void init(BYTE a, BYTE o){
    am=a;
    ot=o;
  }
}TType;
```

Buňka tabulky uchová typ instrukce (`type` - normální, fpu, s rozšířeným operačním kódem). Instrukce obsahuje nula až tři parametry, každý reprezentován typem `TType`. Prvek `mnem` obsahuje jméno instrukce. Pokud položka neobsahuje data je vyplněna speciální hodnotou.

```
typedef struct structOpCodeCell{
  BYTE type;
  const char* mnem;
  TType par1, par2, par3;
}OpCodeCell;
```

Tabulka je konstantním polem,[4] jehož položky jsou typu `OpCodeCell` (buňka). Obdobným způsobem jsou konstruovány a inicializovány ostatní tabulky.

[3] blíže probráno v části Parametry instrukcí 4.1.3

[4] data jsou inicializována při překladu

```
const OpCodeCell OpCode1B[] = {
    OT_NORMAL, ‘‘ADD’’, a_E, o_b, a_G, o_b, _NONE, _NONE,
    OT_NORMAL, ‘‘ADD’’, a_E, o_v, a_G, o_v, _NONE, _NONE,
}
```

5.1.3 Určení přítomnosti ModR/M

Většina instrukcí vyžaduje přítomnost bytu ModR/M. Tyto požadavky jsou dány typem parametrů. Je tedy možné na základě pozice instrukce v tabulce operačních kódů určit výskyt bytu ModR/M. Pro rychlé určení výskytu ModR/M jsem vytvořil bitové pole.

```
//bitové pole pro 1B instrukce
const WORD modRM_1B[]={
  0xF0F0,0xF0F0,0xF0F0,0xF0F0,0x0000,0x0000,0x3050,0x0000,
  0xFFFF,0x0000,0x0000,0x0000,0xCF00,0xF0FF,0x0000,0x0303
};
```

Typ WORD má velikost 16 bitů. 16 buněk má jedna řádka tabulky operačních kódů. Indexem do tabulky (výběrem řádku) jsou čtyři nejvyšší bity (tzv. horní *nibble*). Dolní čtveřice bitů je indexem v rámci zvolené položky pole (index sloupce). Zjištění hodnoty bitu pak získám bitovým posunem a bitovým logickým součinem. Dotaz nad takovou tabulkou pak zajišťuje tato funkce:

```
bool hasModRM_1B(BYTE c){
  return (modRM_1B[c>>4] & (0x8000>>(c&0xF)))!=0;
}
```

Pro tabulku dvoubytových operačních kódů jsem vytvořil podobnou funkci a bitové pole.

5.1.4 Určení velikosti parametru

Kromě několika výjimek (1, 3, EAX) má každý operand velikost danou parametrem v tabulce. Tento samotný parametr neovlivňuje typ operandu. Vliv má pouze na jeho velikost. Je-li operandem adresa paměti, pak parametr určí velikost odkazované paměti (např. `BYTE PTR[400100]`). Dvourozměrné pole obsahuje velikosti parametrů. Parametr určující velikost je indexem do tohoto pole v jedné dimenzi. V druhé dimenzi je indexem nastavený prefix VelikostOperandu.

5.1.5 Dekódování paměti

Velmi častým operandem je ukazatel do paměti. Hodnota prefixu VelikostAdresy ovlivňuje výběr 16-bitové nebo 32-bitové adresace.

Z počátku získáme velikost paměti, na kterou se ukazatel odkazuje. To zajišťují pole `memSize` a `opcodeSize`. První pole obsahuje textové označení jednotlivých velikostí paměti. V poli druhém jsou uloženy velikosti těchto pamětí, které jsou použity jako index do prvního pole.

```
String m_strBuffer+=memSizes[ opcodeSizes[m_bSizePrefix?1:0][size]];
```

Protože adresování paměti u FPU instrukcí má jiné označení (single real, ...). Při dekódování FPU instrukcí je zvolena hodnota, která značí prázdný řetězec. Funkce dekódující FPU instrukci doplní označení paměti ve vlastní režii. Tímto způsobem je využita stejná funkce pro dekódování paměti.

Adresování 16-bitové využívá jednoho či dvou 16-bitových registrů (viz tabulka A.1). Existují tři adresové módy (výběr je závislý na poli Mód bytu ModR/M). Zásadní rozdíl je pouze ve velikosti posunutí, které je v jednotlivých režimech připočteno. V módu `(00)b` není posunutí žádné. Módy `(01)b` a `(10)b` mají 8-bitové a 16-bitové posunutí. Výběr registru nebo páru registrů probíhá jako obvykle. Pomocí tabulky.

```
const char* reg16bAddr[8]={
  ‘‘BX+SI’’,‘‘BX+DI’’,‘‘BP+SI’’,‘‘BP+DI’’,
  ‘‘SI’’,‘‘DI’’,‘‘BP’’,‘‘BX’’
};
```

Jediný rozdíl, kromě posunutí, je v režimu `(00)b`. Zde pro hodnoty, určující registr `BP`, se žádný registr nenachází.

Adresování 32-bitové probíhá poměrně podobným způsobem. Registry jsou vybírány z registrů obecného použití (EAX, EBX, ...). Jednotlivé módy opět značí přítomnost posunutí. Mód `(00)b` je opět bez posunutí. Pro `(01)b` a `(10)b` je posunutí 8-bitové a 32-bitové. Hodnota, která by za normálních okolností značila registr ESP, znamená přítomnost bytu SIB. V režimu `(00)b` je opět drobná odchylka od ostatních. Hodnota, která jinak značící registr EBP, znamená 32-bitové posunutí.

Použitím společných vlastností některých oblastí vstupních dat dosahujeme efektivnějšího dekódování.

5.1.6 SIB

Adresování pomocí bytu SIB vyvolá pouze hodnota bytu ModR/M. Nejprve se určí bázový registr. Pokud je vybrán registr ESP, zjišťuje se hodnota pole Mód bytu ModR/M. Nulová hodnota značí 32-bitové posunutí. Bázovým registrem se stává EBP pro hodnoty `(10)b` a `(11)b`.

Pole index `(100)b` znamená prázdnou hodnotu. Index se nepoužije při výpočtu adresy. Pokud je index použit, pole měřítko zvolí registr, kterým se násobí. Nakonec je načtena hodnota posunutí, pokud byla při dekódování zjištěna.

I zde je použito určování registrů pomocí indexu do pole.

5.1.7 Jména funkcí závislá na prefixu VelikostOperandu

Některé instrukce jsou označovány v závislosti na velikosti operandů. Například instrukce s operačním kódem 0xE3 značí funkci `JECXZ`, kde `EXC` značí 32-bitový registr. Prefix VelikostOperandu mění tuto funkci na `JCXZ`. Ve skutečnosti se jedná o stejnou funkci. Odlišuje se pouze velikostí registru.

Pokud to bylo možné, použil jsem jméno bez označení velikosti. V tabulce operačních kódů jsou vyznačené typické varianty označení instrukce.

U jednobytových instrukcí se jedná o dvojice:

0x98 CBW - CWDE

0x99 CWD - CDQ

0xE3 JCXZ - JECXZ

Podobný jev se vyskytuje také u dvoubytových instrukcí. Zde je situace složitější, protože vliv mají i hodnoty dalších prefixů. Snazším a zřejmě i rychlejším řešením jsou tabulky operačních kódů pro jednotlivé prefixy. Na výběr tabulky má pouze ten poslední.

Některé buňky dvoubytové tabulky ovlivňovány nejsou. Informace o příslušné instrukci se přepíše i do tabulek náležejících jednotlivým prefixům. Tím je zajištěno správné dekódování. Mimo jiné je zachován případný vliv prefixů na dekódování operandů.

5.1.8 Čtení přímých dat

Přímá data jsou součástí instrukce, tudíž jsou konstantní. Jedná se o poslední část kódu instrukce. Proto je nutné zařídit přečtení této hodnoty až po přečtení předchozích částí. Při čtení těchto dat musíme také zohlednit způsob uložení v paměti (tj. pořadí, ve kterém jsou uloženy jednotlivé byty).

5.2 Implementace analyzátoru grafu toku řízení

Jak již bylo v analýze (na straně 26) probráno, je problematika grafu toku řízení rozdělena na dvě části:

- získání grafu toku řízení a
- analýzu grafu toku řízení.

První část byla propojena se získáváním textové reprezentace instrukcí. Bylo nutné udělat drobný zásah do tabulek operačních kódů. Dále bylo nutné rozšířit některé funkce, aby během své činnosti sbíraly informace, které se využijí při určení rozsahu základních bloků. Druhá část analyzuje graf jeho redukováním.

5.2.1 Detekce základních bloků a vytvoření grafu toku řízení

Při detekci bloků využívám informací, které vznikly během získání textové reprezentace instrukcí.

Třída reprezentující základní blok je definována následovně:

```
class bb
{
public:
   bb(BlockTypeBB typ,DWORD beg,DWORD out_size,DWORD out1=0,DWORD out2=0);
   ~bb(void);
   //typ bloku
   BlockTypeBB type;
   //pocatecni adresa
   DWORD begin;
   //pocet vstupu
   DWORD in;
   //pocet vystupu
   DWORD out;
   //vystupy
   DWORD output[2];

   //zvysi pocet vstupu
   void addIn();
   //odstrani jeden vstup
   void removeIn();
   //seznam bloku, ktere tvori tento blok
   vector<bb*> blocks;

};
```

Každý blok je určitého typu — základní blok, ... Každá detekovaná struktura má své označení. Toto označení je obsaženo v položce `type`. Položka `begin` obsahuje adresu počátku základního bloku. Dále obsahuje informace o počtu vstupních hran a počtu výstupních. Včetně jejich cílů. Adresy začátků bloků jsou použity jako označení příslušných bloků. Funkce `addIn` a `removeIn` zajišťují přidávání a odebírání počtu vstupů. Vektor `blocks` obsahuje seznam bloků, ze kterých se blok skládá.

Pro snadnější vyhledávání jsou ukazatele na instance bloků uloženy v mapě. Klíčem je počáteční adresa bloku. Během ustavení grafu toku řízení vytvářím pro každý blok množinu předchůdců a přímých předchůdců. Tyto informace jsou následně využity k další analýze.

5.2.2 Detekce základních struktur

Detekování struktur je procházení seznamem základních bloků a testování na určité podmínky (jediná vstupní hrana, dvě výstupní hrany, ...). Blíže bylo probráno v analýze problému (26).

Výstupy analyzátoru jsou ve formátu XML. Tento soubor obsahuje informace potřebné ke znovusestavení grafu toku řízení bez přítomnosti analyzovaného programu.

Kapitola 6

Závěr

Výsledkem této práce je program, jehož výstupem je textová podoba analyzovaného programu v jazyce symbolických instrukcí a detekované struktury ve formátu XML.

Program generuje dobré výsledky, protože při porovnání výstupů vlastního programu s výstupy referenčního programu dumpbin jsem pozoroval rozdíly pouze ve formátování výstupu. Sada testovaných programů a výstupů (textová podoba instrukcí a detekované struktury) je přiložena na CD.

Kvůli možnostem zpětné vazby jazyka symbolických instrukcí není možné získat statickou analýzou graf toku řízení z libovolného EXE souboru. Lze získat jen jeho omezenou část. Aby bylo možné detekovat nějaké struktury, program problém zpětné vazby neuvažuje. Možnosti detekce jsou tedy omezené.

Analyzátor grafu toku řízení dokáže detekovat základní konstrukce: if-then, if-then-else, while-do, do-while. Vypisuje seznam bloků, ze kterých bloků se skládají. Rozeznává pouze jednoduché podgrafy, které redukuje. Složitější konstrukce zkráceně vyhodnocovaných podmínek detekovat nedokáže.

Program by bylo možné rozšířit o detekování funkce `main`. V této oblasti má řada disasemblerů špatné výsledky. Další možností rozšíření je detekce složitějších struktur s využitím dalších postupů z oblasti formálních jazyků a překladačů.

Nové technologie 64-bitových počítačů si vyžádají analyzátory 64-bitových programů. Bude nutné rozšířit disasembler o funkce analyzující tyto aplikace.

Možnost využití tohoto projektu je v současné době malá, protože jeho schopnosti detekce jsou omezené. Do budoucna po rozšíření funkčnosti jej lze využít k získání charakteristických znaků programu a detekci podobného kódu. Zde jsou využití v oblasti počítačové bezpečnosti veliké.

Literatura

[1] Ida Pro Disassembler and Debugger. http://www.datarescue.com/idabase/.

[2] OllyDbg 32-bit Assembler-Level Debugger. http://www.ollydbg.de/.

[3] C. Cifuentes. Reverse compilation techniques. http://citeseer.comp.nus.edu.sg/cifuentes94reverse.html, 1994.

[4] A. Meduna. Www stránky. http://www.fit.vutbr.cz/study/courses/VYP/.

[5] Tutoriál na WWW. The Art of Disassembly. http://www.nuc.cz/tutorials/, 2003. [dostupné srpen 2006].

[6] Www stránky. Microsoft Portable Executable and Common Object File Format Specification. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

[7] Peter Szor. *Počítačové viry : analýza a obrana*. Zonner Press, 2006. ISBN 80-86815-04-8.

[8] WWW. Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/design/processor/manuals/, 2006. [dostupné srpen 2006].

Seznam příloh

Dodatek A

Přílohy

A.1 Ukázka výstupu vlastního disasembleru

```
00401000: 55                  PUSH  EBP
00401001: 89E5                 MOV  EBP,ESP
00401003: 83EC18               SUB  ESP,18
00401006: 895DF8               MOV  DWORD PTR [EBP-8],EBX
00401009: 8B5508               MOV  EDX,DWORD PTR [EBP+8]
0040100C: 31DB                 XOR  EBX,EBX
0040100E: 8975FC               MOV  DWORD PTR [EBP-4],ESI
00401011: 8B02                 MOV  EAX,DWORD PTR [EDX]
00401013: 31F6                 XOR  ESI,ESI
00401015: 8B00                 MOV  EAX,DWORD PTR [EAX]
00401017: 3D910000C0           CMP  EAX,C0000091
0040101C: 7743                  JA  00401061
0040101E: 3D8D0000C0           CMP  EAX,C000008D
00401023: 725B                  JB  00401080
00401025: BE01000000           MOV  ESI,00000001
0040102A: C70424080000         MOV  DWORD PTR [ESP],00000008
          00
00401031: 31C0                 XOR  EAX,EAX
00401033: 89442404             MOV  DWORD PTR [ESP+4],EAX
00401037: E8A4070000          CALL  004017E0
0040103C: 83F801               CMP  EAX,01
0040103F: 746C                  JE  004010AD
00401041: 85C0                TEST  EAX,EAX
00401043: 742A                  JE  0040106F
00401045: C70424080000         MOV  DWORD PTR [ESP],00000008
          00
0040104C: FFD0                CALL  EAX
0040104E: BBFFFFFFFF           MOV  EBX,FFFFFFFF
00401053: 89D8                 MOV  EAX,EBX
00401055: 8B75FC               MOV  ESI,DWORD PTR [EBP-4]
00401058: 8B5DF8               MOV  EBX,DWORD PTR [EBP-8]
0040105B: 89EC                 MOV  ESP,EBP
0040105D: 5D                   POP  EBP
```

A.2 Výstup programu dumpbin

```
00401000: 55                push        ebp
00401001: 89 E5             mov         ebp,esp
00401003: 83 EC 18          sub         esp,18h
00401006: 89 5D F8          mov         dword ptr [ebp-8],ebx
00401009: 8B 55 08          mov         edx,dword ptr [ebp+8]
0040100C: 31 DB             xor         ebx,ebx
0040100E: 89 75 FC          mov         dword ptr [ebp-4],esi
00401011: 8B 02             mov         eax,dword ptr [edx]
00401013: 31 F6             xor         esi,esi
00401015: 8B 00             mov         eax,dword ptr [eax]
00401017: 3D 91 00 00 C0    cmp         eax,0C0000091h
0040101C: 77 43             ja          00401061
0040101E: 3D 8D 00 00 C0    cmp         eax,0C000008Dh
00401023: 72 5B             jb          00401080
00401025: BE 01 00 00 00    mov         esi,1
0040102A: C7 04 24 08 00 00 mov         dword ptr [esp],8
          00
00401031: 31 C0             xor         eax,eax
00401033: 89 44 24 04       mov         dword ptr [esp+4],eax
00401037: E8 A4 07 00 00    call        004017E0
0040103C: 83 F8 01          cmp         eax,1
0040103F: 74 6C             je          004010AD
00401041: 85 C0             test        eax,eax
00401043: 74 2A             je          0040106F
00401045: C7 04 24 08 00 00 mov         dword ptr [esp],8
          00
0040104C: FF D0             call        eax
0040104E: BB FF FF FF FF    mov         ebx,0FFFFFFFFh
00401053: 89 D8             mov         eax,ebx
00401055: 8B 75 FC          mov         esi,dword ptr [ebp-4]
00401058: 8B 5D F8          mov         ebx,dword ptr [ebp-8]
0040105B: 89 EC             mov         esp,ebp
0040105D: 5D                pop         ebp
```

A.3 Tabulky

Tabulka A.1: 16 a 32 bitové adresové módy s bytem ModR/M

Mod	R/M	16-bitový adresový mód	32-bitový adresový mód
00	000	[BX+SI]	[EAX]
	001	[BX+DI]	[ECX]
	010	[BP+SI]	[EDX]
	011	[BP+DI]	[EBX]
	100	[SI]	[-][- -][1]
	101	[DI]	disp32
	110	disp16	[ESI]
	111	[BX]	[EDI]
01	000	[BX+SI] + disp8	[EAX] + disp8
	001	[BX+DI] + disp8	[ECX] + disp8
	010	[BP+SI] + disp8	[EDX] + disp8
	011	[BP+DI] + disp8	[EBX] + disp8
	100	[SI] + disp8	[-][- -][1] + disp8
	101	[DI] + disp8	[EBP] + disp8
	110	[BP] + disp8	[ESI] + disp8
	111	[BX] + disp8	[EDI] + disp8
10	000	[BX+SI] + disp16	[EAX] + disp32
	001	[BX+DI] + disp16	[ECX] + disp32
	010	[BP+SI] + disp16	[EDX] + disp32
	011	[BP+DI] + disp16	[EBX] + disp32
	100	[SI] + disp16	[-][- -][1]+ disp32
	101	[DI] + disp16	[EBP] + disp32
	110	[BP] + disp16	[ESI] + disp32
	111	[BX] + disp16	[EDI] + disp32

Tabulka A.2: kódy registrů

	Obecné registry			MMX	XMM	Kontrolní	Ladicí
	8b	16b	32b				
000	AL	AX	EAX	MM0	XMM0	CR0	DR0
001	CL	CX	ECX	MM1	XMM1	rezervované	DR1
010	DL	DX	EDX	MM2	XMM2	CR2	DR2
011	BL	BX	EBX	MM3	XMM3	CR3	DR3
100	AH	SP	ESP	MM4	XMM4	CR4	rezervované
101	CH	BP	EBP	MM5	XMM5	rezervované	rezervované
110	DH	SI	ESI	MM6	XMM6	rezervované	DR6
111	BH	DI	EDI	MM7	XMM7	rezervované	DR7