UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Sašo Pajntar

Onemogočanje obratnega inženiringa strojne kode

DIPLOMSKO DELO
NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Mentor: doc. dr. Peter Peer

Ljubljana, 2011

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: 00159/2011
Datum: 06.09.2011

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

Kandidat: **SAŠO PAJNTAR**

Naslov: **ONEMOGOČANJE OBRATNEGA INŽENIRINGA STROJNE KODE**
THWARTING REVERSE ENGINEERING OF MACHINE CODE

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Opišite teorijo, ki je potrebna za razumevanje obratnega inženiringa strojne kode. Nato predstavite orodja za izvajanje obratnega inženiringa, delovanje le-teh ter metode za onemogočanje obratnega inženiringa. Izdelajte orodje za onemogočanje obratnega inženiringa, ki bo temeljilo na uporabi odvečne in samoreplikacijske kode. Na koncu predstavite uporabo razvitega orodja za zaščito strojne kode pred obratnim inženiringom, njegove slabosti ter možne izboljšave.

Mentor:
doc. dr. Peter Peer

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Dekan:
prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Sašo Pajntar,

z vpisno številko 63030259,

sem avtor diplomskega dela z naslovom:

Onemogočanje obratnega inženiringa strojne kode

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  doc. dr. Petra Peera,
- 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 15. 12. 2011 Podpis avtorja:

Zahvala

Zahvaljujem se doc. dr. Petru Peeru za strokovno pomoč pri izvedbi diplomske naloge.

Kazalo

Povzetek **1**

Abstract **2**

1 Uvod **3**

2 Obratni inženiring **5**
2.1 Format PE 5
2.1.1 Glava DOS 6
2.1.2 Glava PE 8
2.1.3 Podatkovni imenik 10
2.1.4 Tabela sekcij 11
2.1.5 Uvozna sekcija 13
2.1.6 Izvozna sekcija 15
2.2 Zbirni jezik x86 16
2.2.1 Načini izvajanja zbirnika x86 17
2.2.2 Registri 19
2.2.3 Format zbirnega jezika 30
2.3 Vprašanje legalnosti obratnega inženiringa 34

3 Orodja za izvajanje obratnega inženiringa **36**
3.1 Sistemski monitorji 37
3.2 Razhroščevalniki 38
3.2.1 Razhroščevalniki v uporabniškem načinu 39
3.2.2 Razhroščevalniki v sistemskem načinu 41
3.3 Obratni zbirniki 43
3.3.1 Linearni algoritem 44
3.3.2 Rekurzivni algoritem 45

4 Metode nasprotno-obratnega inženiringa **47**
4.1 Metode za zaznavanje monitorjev 48
4.1.1 Zaznavanje z iskanjem gonilnikov 48
4.1.2 Zaznavanje z iskanjem registriranega razreda 49
4.2 Metode za zaznavanje razhroščevalnikov 49
4.2.1 Zaznavanje s pomočjo API klicev 50
4.2.2 Zaznavanje prekinitvenih točk 51
4.2.3 Zaznavanje s časovno razliko 53
4.3 Metode za onemogočanje analize obratnega zbirnika 54
4.3.1 Odvečna koda 55
4.3.2 Samoreplikacijska koda 57
4.3.3 Uporaba navideznaga stroja 58
4.4 Primerjava 59
4.4.1 Upx 3.07 63
4.4.2 ASProtect 1.64 65
4.4.3 PeLock 1.06 69
4.4.4 Enigma Protector 3.10 71

5 Odvečna koda **75**
5.1 Metoda s pogojnimi in nepogojnimi skoki 76
5.2 Metoda s klicem podprograma 78

6 Samoreplikacijska koda **81**
6.1 Polimorfizem 83
6.2 Metamorfizem 86

7 Primer uporabe **92**
7.1 Razvojno-programski paket 92
7.2 Generator samoreplikacijske kode 97
7.3 Rezultati 100
7.3.1 Primerjava 108
7.4 Možne izboljšave 109

8 Zaključek **111**

Seznam slik **112**
Seznam tabel **114**

Viri **115**

Seznam uporabljenih kratic in simbolov

EXE (angl. executable file) – izvajalna datoteka

DLL (angl. dynamic link library) – dinamična knjižnica

PE format (angl. portable executable format) – format izvajalnih datotek

SDK (angl. software development kit) – razvojno-programski paket

GPF (angl. general protection fault) – splošno zaščitna izjema, sproži se v procesorju ob prekoračitvi zaščite

API (angl. application programming interface) – vmesnik, ki zagotavlja dostop do funkcij operacijskega sistema ali drugega računalniškega programa

SEH (angl. structured exception handling) – strukturirana obravnava izjem

Seznam pojmov

Hrošč (angl. bug) – napaka v programu, ki omogoča nepredvidljivo delovanje programske opreme.

Razhroščevalnik (angl. debugger) – orodje, ki omogoča kontrolirano izvajanje programov. Z njim lahko spreminjamo vrednosti registrov, vrednosti v pomnilniku ter zbirno kodo.

Obratni zbirnik (angl. disassembler) – orodje, ki strojno kodo prevede v zbirni jezik.

Pakirnik (angl. packer) – orodje, s katerim skrčimo EXE, DLL ali katero koli drugo PE datoteko.

Povzetek

Obratni inženiring je proces, pri katerem skušamo odkriti razvojne podrobnosti programske opreme, pri čemer nimamo dostopa do izvorne kode programa. V diplomski nalogi smo predstavili proces obratnega inženiringa v operacijskem sistemu Microsoft Windows. Najprej smo podrobno opisali PE format ter zgradbo zbirnika x86. Sledi opis orodij, ki se uporabljajo v procesu obratnega inženiringa. Pri orodjih smo se osredotočili na razhroščevalnike, obratne zbirnike ter sistemske monitorje. Predstavljeno je delovanje orodij ter metode, ki služijo za zaznavanje ter onemogočanje izvajanja obratnega inženiringa. Osredotočili smo se na uporabo odvečne in samoreplikacijske kode, saj je bil cilj diplomske naloge izdelati orodje, ki bo s pomočjo teh metod kodiralo in dekodiralo strojno kodo. Na koncu smo predstavili uporabo orodja, njegove slabosti ter možne izboljšave, ki bi pripomogle k boljši zaščiti strojne kode.

Ključne besede:

format PE, zbirnik x86, razhroščevalnik, obratni zbirnik, sistemski monitor, odvečna koda, samoreplikacijska koda

Abstract

Reverse engineering is a process in which we try to discover the details of software, while we do not have access to its source code. In the diploma thesis we present reverse engineering process on Microsoft Windows operating system. First, we describe PE format and structure of x86 assembler. Then we describe tools, which are used in process of reverse engineering. The focus is on debuggers, dissasemblers and system monitors. We describe functionality of such tools to detect and thwart reverse engineering process. We focus on the use of junk and self-modifying code, since the goal of the thesis was to develope a tool, which will use these methods to code and decode machine instructions. Finaly, we describe the use of the tool, its weaknesses and possible improvements for better protection of machine code.

Key words:

PE format, x86 assembler, debugger, dissasembler, system monitor, junk code, self-modifying code

Poglavje 1

Uvod

Računalnik je programabilna elektronska naprava, ki omogoča samodejno izvajanje določenega zaporedja aritmetičnih ter logičnih ukazov. Ti ukazi so binarno zaporedje ničel in enic, ki predstavljajo strojno kodo, ki jo izvaja centralno procesna enota. Računalnik razume strojni jezik, je pa precej manj razumljiv za človeka. Zato so začeli razvijati programske jezike, ki so za človeka razumljivejši, potrebujejo pa prevajalnik za prevod izvorne kode v strojno. Nižji programski jezik je zagotovo zbirni jezik, ki je interpretacija strojnega jezika v človeku razumljivejše ukaze (slika 1.1).

```
; Example of IBM PC assembly language
; Accepts a number in register AX;
; subtracts 32 if it is in the range 97-122;
; otherwise leaves it unchanged.

SUB32  PROC         ; procedure begins here
       CMP  AX,97   ; compare AX to 97
       JL   DONE    ; if less, jump to DONE
       CMP  AX,122  ; compare AX to 122
       JG   DONE    ; if greater, jump to DONE
       SUB  AX,32   ; subtract 32 from AX
DONE:  RET          ; return to main program
SUB32  ENDP         ; procedure ends here
```

Slika 1.1: Procedura napisana v zbirnem jeziku

Danes se programiranje v zbirniku ne uporablja več dosti, saj je razvoj programske opreme v zbirniku časovno precej zahteven. Programer mora natančno poznati tudi zgradbo in delovanje procesne enote, saj je zbirnik neposredno odvisen od nje. Zbirnik so nadomestili razvitejši programski jeziki kot so C, C++, Object Pascal, Java in še mnogi drugi. Eni uporabljajo sprotni prevajalnik za prevajanje programske kode v vmesno kodo (C#, Java), drugi kodo optimizirajo in jo prevedejo v strojno kodo (slika 1.2).

Slika 1.2: Uporaba visokonivojskih jezikov

Rezultat prevajalnika je optimizirana strojna koda, ki se lahko izvaja na računalniški platformi, za katero je bila napisana. V kolikor imamo izvorno kodo, lahko enostavno s pomočjo prevajalnika izdelamo izvršljivo datoteko. Do programske kode pa se je težje dokopati, ko imamo samo izvršljivo datoteko. Strojno kodo se najlažje pretvori v zbirni jezik, kjer se analizira samo delovanje programa. Analizo delovanja programa in razkrivanje razvojnih zakonitosti programa imenujemo obratni inženiring.

V diplomski nalogi smo se osredotočili predvsem na orodja, ki omogočajo obratni inženiring v operacijskem sistemu Microsoft Windows ter na tehnike, ki otežijo analiziranje strojne kode.

Poglavje 2

Obratni inženiring

Obratni inženiring (angl. reverse engineering) ima za cilj ustvarjanje ekvivalenta originala izvorne kode programa, oziroma poskus odkritja razvojnih podrobnosti programa.

Za dosego tega cilja se uporabljajo različni programi, kot so razhroščevalniki (angl. debuggers), obratni zbirniki (angl. disassemblers) ter sistemski monitorji. Poznati je potrebno tudi zbirni jezik, saj razhroščevalniki in obratni zbirniki strojno kodo pretvorijo v zbirno. Prav tako je priporočljivo poznati format izvršljivih datotek, ki jih zgradi prevajalnik, saj vsebuje vse informacije o samem programu. Operacijski sistem Windows uporablja format PE (angl. Portable Executable Format), ki je spremenjena različica Unix-ovega COFF (angl. Common Object File Format) formata, uporablja pa se od leta 1993, ko ga je Microsoft uvedel v os Windows NT 3.1.

2.1 Format PE

Format PE se uporablja za vse izvršljive datoteke, dinamične knjižnice, .NET aplikacije ter gonilnike. Format je sestavljen iz glave DOS (angl. Dos header), glave PE (angl. PE header), tabele sekcij (angl. section table), sledijo še posamezne sekcije [1, 2]. Sekcije, ki se najpogosteje pojavljajo v običajnih PE formatih, so naslednje:

- ".text" ali "CODE" sekcija, vsebuje strojno kodo, ki jo izvaja aplikacija.
- ".rdata" sekcija vsebuje podatke, ki so samo za branje (nizi, ki so v aplikaciji uporabljeni, konstante ter podatki).
- ".bss" sekcija vsebuje neinicializirane podatke in statične spremenljivke.

- ".data" sekcija vsebuje inicializirane podatke, ki jih aplikacija uporablja kot globalne spremenljivke.
- ".rsrc" sekcija vsebuje podatke, kot so aplikacijski dialogi, ikone, nizi.
- ".edata" sekcija vsebuje izvozni direktorij (angl. export directory), vsebuje imena in naslove funkcij, ki druge aplikacije kličejo. Ta sekcija se največ uporablja v dinamičnih knjižnicah, imenujemo jo tudi izvozna sekcija.
- ".idata" ali uvozna sekcija vsebuje različne informacije o uvoženih funkcijah, ki jih aplikacija kliče pri svojem izvajanju.

Nalagalnik (angl. loader) naloži format PE v pomnilnik, razreši uvožene API (angl. application programming interface) naslove ter dodeli sklad in kopico (angl. heap). Format se zaporedno preslika v pomnilnik, razlika je le pri poravnani sekcij (angl. section alignment). Sekcije so navadno poravnane na 4096 besed (angl. byte) oziroma na vrednost, ki je določena v PE formatu.

2.1.1 Glava DOS

Prvih 64 besed PE formata se začne z DOS glavo (izpis 2.1). Namenjena je kompatibilnosti z operacijskim sistemom MS-DOS. Zaradi te glave MS-DOS prepozna PE format kot veljavno datoteko za izvajanje in izvede kratek 16-bitni program, ki sledi glavi DOS in navadno samo izpiše opozorilo, da je potrebno program izvesti v okolju Windows. Glava se začne s poljem, ki vsebuje besedi šestnajstiške vrednosti 0x4D,0x5A (niz "MZ"), sledijo še ostala polja. Omeniti velja zadnje polje imenovano "e_lfanew", ki vsebuje odmik v datoteki, kjer se nahaja začetek glave PE. Izpis 2.1 prikazuje strukturo DOS glave [2]. DOS glavi sledi 16-bitni program (izpis 2.2), ki se izvede, če poženemo datoteko v MS-DOS sistemu.

```
typedef struct _IMAGE_DOS_HEADER
{
  WORD e_magic;
  WORD e_cblp;
  WORD e_cp;
  WORD e_crlc;
  WORD e_cparhdr;
  WORD e_minalloc;
```

```
    WORD e_maxalloc;
    WORD e_ss;
    WORD e_sp;
    WORD e_csum;
    WORD e_ip;
    WORD e_cs;
    WORD e_lfarlc;
    WORD e_ovno;
    WORD e_res[4];
    WORD e_oemid;
    WORD e_oeminfo;
    WORD e_res2[10];
    LONG e_lfanew;
  }
  IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Izpis 2.1: Struktura DOS glave

Program kliče programske prekinitve (angl. software interrupts). To so: API klici do DOS sistemskih rutin. Vsaka prekinitev ima številko, ki pomeni odmik v tabeli prekinitvenih vektorjev [3]. Program kliče vektor na odmiku 0x21, vrednost v AH registru pa služi kot parameter, ki določi, katero podfunkcijo bo prekinitev izvedla. V izpisu 2.2 se najprej uporabi parameter 0x09. Prekinitev izvede podfunkcijo, ki izpiše na zaslon niz, kazalec na niz pa se nahaja v DS in DX registrih. Prvi register vsebuje začetni naslov segmenta, drugi pa odmik v segmentu, kjer se niz nahaja. Drugi klic prekinitve uporablja parameter 0x4C, ki program prekine. Register AL vsebuje kodo napake, ki bo posredovana sistemu [4].

```
0X0000 PUSH CS
0X0001 POP DS
0X0002 MOV DX,0X000E
0X0005 MOV AH,0X09
0X0007 INT 0X21
0X0009 MOV AX,0X4C01 //AH = 0X4C, AL = 0X01
0X000C INT 0X21
0x000E "This program can not be run in dos mode$"
```

Izpis 2.2: 16-bitni program iz programa Notepad.exe

2.1.2 Glava PE

Glava PE (izpis 2.3) se začne v datoteki na odmiku, ki ga vsebuje "e_lfanew" polje v DOS glavi. Sestavljena je iz podpisa (angl. PE signature), datotečne glave (angl. file header) in neobvezne glave (angl. optional header), ki pa je vedno prisotna v PE glavi. Podpis je sestavljen iz besed 0x50, 0x45h, 0x00, 0x00 (niz "PE" z dvema *"null"* besedama).

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

Izpis 2.3: Struktura PE glave

Datotečna glava (izpis 2.4) vsebuje informacije o sistemu, število sekcij, časovni žig, kazalec na tabelo simbolov, število simbolov, velikost neobvezne glave ter karakteristike PE glave. Karakteristike definirajo, ali je datoteka dinamična knjižnica, .NET aplikacija ali katera druga izvajalna datoteka.

```
typedef struct _IMAGE_FILE_HEADER
{
  WORD Machine;
  WORD NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD SizeOfOptionalHeader;
  WORD Characteristics;
}
IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Izpis 2.4: Struktura datotečne glave

Neobvezna glava (izpis 2.5) je struktura v velikost 224 besed. Vsebuje naslove za preslikavo PE formata v pomnilnik, naslov vstopne točke programa (angl. program entry point) ter nekaj drugih podatkov. Pomembnejša polja so naslednja:

- "AddressOfEntryPoint" polje vsebuje relativni naslov, kjer se bo nahajal začetek programa.
- "ImageBase" polje vsebuje osnovni naslov, kamor se bo PE format preslikal v pomnilnik.
- "SectionAlignment" vsebuje vrednost, na katero so poravnane sekcije, ki se naložijo v pomnilnik.
- "SizeOfImage" vsebuje velikost PE formata, ki se bo preslikal v pomnilnk.
- "CheckSum" vsebuje kontrolni seštevek PE formata.
- "NumberOfRvaAndSizes" vsebuje število podatkovnih imenikov.
- "DataDirectory" je kazalec na prvi podatkovni imenik.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD Magic;
    BYTE MajorLinkerVersion;
    BYTE MinorLinkerVersion;
    DWORD SizeOfCode;
    DWORD SizeOfInitializedData;
    DWORD SizeOfUninitializedData;
    DWORD AddressOfEntryPoint;
    DWORD BaseOfCode;
    DWORD BaseOfData;
    DWORD ImageBase;
    DWORD SectionAlignment;
    DWORD FileAlignment;
    WORD MajorOperatingSystemVersion;
    WORD MinorOperatingSystemVersion;
    WORD MajorImageVersion;
    WORD MinorImageVersion;
    WORD MajorSubsystemVersion;
    WORD MinorSubsystemVersion;
    DWORD Win32VersionValue;
    DWORD SizeOfImage;
    DWORD SizeOfHeaders;
    DWORD CheckSum;
    WORD Subsystem;
```

```
    WORD DllCharacteristics;
    DWORD SizeOfStackReserve;
    DWORD SizeOfStackCommit;
    DWORD SizeOfHeapReserve;
    DWORD SizeOfHeapCommit;
    DWORD LoaderFlags;
    DWORD NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[16];
}
IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Izpis 2.5: Struktura neobvezne glave

2.1.3 Podatkovni imenik

Podatkovni imenik je tabela 16 struktur (izpis 2.6), nahaja pa se na koncu neobvezne glave (polje "DataDirectory" v izpisu 2.5). Vsako strukturo sestavljata dve polji. Polje "VirtualAddress" vsebuje relativni navidezni odmik, kjer se nahaja nova struktura, ki vsebuje pomembne podatke za izvajalno datoteko. Polje "Size" vsebuje velikost same strukture v besedah.

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD VirtualAddress;
    DWORD Size;
}
IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Izpis 2.6: Struktura elementa podatkovnega imenika

Vsaka struktura, na katero kažejo elementi podatkovnega imenika vsebuje različne podatke, ki so pomembni za sam zagon izvajalne datoteke. Tako podatkovni imenik vsebuje informacije o uvozni in izvozni tabeli, tabeli virov (angl. resource table), varnostni tabeli, CLR glavi (angl. common language runtime header) in še nekatere druge. Strukture v podatkovnem imeniku si sledijo tako, kot prikazuje tabela 2.1.

Indeks podatkovne tabele	**Tip podatkovne strukture**
0	Izvozna tabela
1	Uvozna tabela

2	Tabela virov
3	Tabela izjem
4	Varnostna tabela
5	Osnovna premestitvena tabela
6	Informacije za razhroščevalnike
7	Arhitekturne informacije
8	Relativni navidezni naslov na register, ki vsebuje kazalec na globalne spremenljivke
9	Kazalec na lokalni pomnilnik niti
10	Tabela konfiguracij
11	Robna uvozna tabela
12	Naslov uvozne tabele
13	Uvozni deskriptor
14	Glava CLR
15	Rezervirano za prihodnjo uporabo

Tabela 2.1: Tabela podatkovnega imenika

2.1.4 Tabela sekcij

Tabela sekcij je sestavljena iz sekcijskih glav (izpis 2.7), ki vsebujejo podatke o določeni sekciji v PE formatu. Število sekcij vsebovanih v PE formatu določa polje "NumberOfSections" v datotečni glavi (izpis 2.4).

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE Name[8];
  union {
          DWORD PhysicalAddress;
          DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
```

```
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
  }
  IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Izpis 2.7: Struktura sekcijske glave

Posamezna polja imajo naslednje pomene:

- "Name" vsebuje ime sekcije, ki je omejeno z dolžino 8 znakov.
- "Misc" je unija dveh polj, "PhysicalAddress" in "VirtualSize". "PhysicalAddress" vsebuje fizični odmik sekcije v datoteki, "VirtualSize" pa vsebuje velikost naložene sekcije v pomnilniku.
- "VirtualAddress" vsebuje relativni navidezni naslov, kam v pomnilnik naj se sekcija preslika (primer: če polje vsebuje vrednost 0x2000 ter je polje "ImageBase" v neobvezni glavi nastavljeno na 0x00400000, potem se bo sekcija v pomnilnik naložila na naslov 0x00402000).
- "SizeOfRawData" vsebuje velikost sekcije v datoteki, ki je zaokrožena glede na vrednost polja "FileAlignment" v neobvezni glavi (izpis 2.5).
- "PointerToRawData" vsebuje absolutni odmik, kje v datoteki se nahaja sekcija.
- "PointerToRelocations" vsebuje fizični odmik do prestavljenih vnosov (angl. relocation entries) sekcije. Če prestavljenih vnosov ni, je polje nastavljeno na vrednost 0.
- "PointerToLinenumbers" vsebuje fizični odmik do vnosa, ki vsebuje številke vrstic programske kode za trenutno sekcijo. Če ni informacij o številkah vrstic, potem je polje nastavljeno na 0.
- "NumberOfRelocations" vsebuje število prestavljenih vnosov.
- "NumberOfLinenumbers" vsebuje število vnosov, ki vsebujejo številke vrstic programske kode.
- "Characteristics" polje vsebuje zastavice (angl. flags), ki določajo parametre sekcije. Označujejo, ali sekcija vsebuje kodo, inicializirane ali neinicializirane podatke, je sekcija označena samo za branje in podobno.

Tipične sekcije, ki jih vsebujejo izvajalne datoteke ali dinamične knjižnice, so uvozna in izvozna sekcija, sekcija virov, sekcija s strojno kodo in sekcija z inicializiranimi podatki. Nekatere aplikacije nimajo posebej definirane izvozne ali uvozne sekcije, zato se uvozna ali izvozna tabela iz podatkovnega imenika (tabela 2.1) nahaja v podatkovni sekciji. Kljub temu smo obe sekciji v nadaljevanju opisali, saj vsebujeta uvozno in izvozno tabelo, ki sta pomembna dela PE formata.

2.1.5 Uvozna sekcija

Uvozna sekcija vsebuje informacije glede vseh Windows API klicev, ki jih aplikacija uporablja za svoje delovanje. Sekcija vsebuje uvozno tabelo, element tabele predstavlja struktura "IMAGE_IMPORT_DESCRIPTOR" (izpis 2.8). Vsaka struktura vsebuje informacije o posamezni dinamičnih knjižnici, ki jo aplikacija potrebuje za svoje izvajanje. Struktura, ki ima vsa polja postavljena na 0, označuje zadnji element v uvozni tabeli.

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
  union {
      DWORD Characteristics;
      DWORD OriginalFirstThunk;
  };
  DWORD TimeDateStamp;
  DWORD ForwarderChain;
  DWORD Name;
  DWORD FirstThunk;
}
IMAGE_IMPORT_DESCRIPTOR;
```

Izpis 2.8: Struktura elementa uvozne tabele

Pomen polj je naslednji:

- "OriginalFirstThunk" polje je unija s poljem "Characteristics", ki naj bi vsebovala parametre, vendar se vedno uporablja samo polje "OriginalFirstThunk". Vsebuje relativni navidezni naslov do tabele struktur (izpis 2.9), ki vsebuje podatke o uvoženih funkcijah.

- "TimeDateStamp" polje vsebuje časovni žig prevajalnika, vendar je največkrat nastavljeno na vrednost 0.

- "ForwarderChain" polje se ne uporablja in ima vrednost nastavljeno na -1 ali 0.
- "Name" vsebuje relativni navidezni naslov, kjer se nahaja ime uvožene dinamične knjižnice.
- "FirstThunk" polje vsebuje relativni navidezni naslov do kopije tabele, na katero kaže polje "OriginalFirstThunk".

```
typedef struct _IMAGE_THUNK_DATA32 {
  union {
        DWORD ForwarderString;
        DWORD Function;
        DWORD Ordinal;
        DWORD AddressOfData;
  } u1;
}
IMAGE_THUNK_DATA32;
```

Izpis 2.9: Struktura IMAGE_THUNK_DATA32

Struktura "IMAGE_THUNK_DATA32" je unija štirih polj, od katerih pa se uporabljata samo dve, "Ordinal" ali "AddressOfData". Če se funkcija, ki jo aplikacija uporablja, uvaža po identifikacijski številki, potem polje "Ordinal" vsebuje identifikacijsko številko uvožene funkcije, če pa se uvaža z imenom, polje "AddressOfData" vsebuje relativni navidezni naslov do strukture (izpis 2.10), ki vsebuje ime uvožene funkcije:

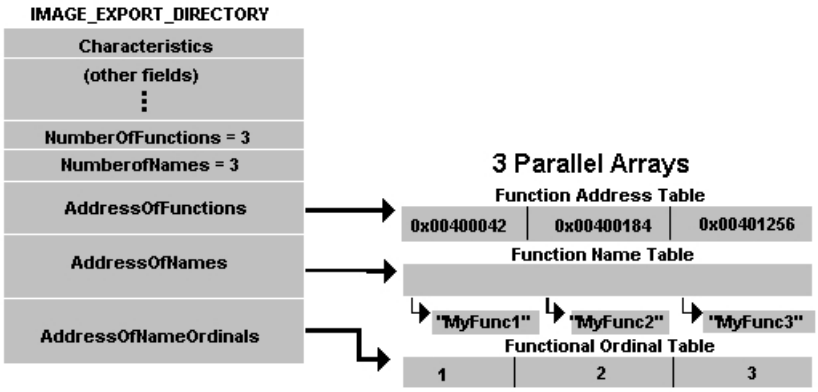
```
typedef struct _IMAGE_IMPORT_BY_NAME {
  WORD Hint;
  BYTE Name[1];
}
IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

Izpis 2.10: Struktura IMAGE_IMPORT_BY_NAME

- "Hint" polje vsebuje indeks na izvozno tabelo v dinamični knjižnici, kjer se ta funkcija nahaja. Uporablja ga nalagalnik pri preslikavi uvoznih funkcij.
- "Name" polje je dinamična tabela, ki vsebuje ime uvožene funkcije.

2.1.6 Izvozna sekcija

Izvozna sekcija vsebuje informacije o izvoznih funkcijah, ki jih aplikacije uporabljajo za svoje delovanje. Informacije so shranjene v izvozni tabeli (slika 2.1).



Slika 2.1: Izvozna tabela

Funkcije se lahko izvozijo po imenu ali po identifikacijski številki. Identifikacijska številka je 16-bitna in je unikatna samo znotraj posamezne dinamične knjižnice. Strukturo tabele opisuje struktura "IMAGE_EXPORT_DIRECTORY" (izpis 2.11), ki je sestavljena iz enajstih polj, od katerih pa se uporabljajo le nekatera, opisana v nadaljevanju:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
  DWORD Characteristics;
  DWORD TimeDateStamp;
  WORD MajorVersion;
  WORD MinorVersion;
  DWORD Name;
  DWORD Base;
  DWORD NumberOfFunctions;
  DWORD NumberOfNames;
  DWORD AddressOfFunctions;
  DWORD AddressOfNames;
  DWORD AddressOfNameOrdinals;
}
IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Izpis 2.11: Struktura izvozne tabele

- "Name" vsebuje kazalec na interno ime dinamične knjižnice.
- "Base" vsebuje začetno identifikacijsko številko izvoznih funkcij.
- "NumberOfFunctions" vsebuje število vseh izvoznih funkcij.
- "NumberOfNames" vsebuje število funkcij, ki so izvožene po njihovih imenih.
- "AddressOfFunctions" vsebuje relativni navidezni naslov na tabelo izvoznih naslovov funkcij (angl. export address table).
- "AddressOfNames" vsebuje relativni navidezni naslov na tabelo izvoznih imen funkcij (angl. export name table).
- "AddressOfNameOrdinals" vsebuje reliativni navidezni naslov na tabelo izvoznih identifikacijskih številk (angl. export ordinal table).

2.2 Zbirni jezik x86

Za izvajanje obratnega inženiringa je poleg poznavanja PE formata potrebno poznati zbirni jezik. Zbirni jezik x86 je družina zbirnih jezikov, ki so se pojavljali skozi zgodovino razvoja Intelovih procesorjev. Zbirnik x86 pozna dve vrsti sintakse, Intel-ovo in sintakso AT&T. Glavne razlike med obema sintaksama so opisane v tabeli 2.2. Večina zbirnikov, kot so MASM (Microsoft Macro Assembler), TASM (Borland Turbo Assembler), NASM (Netwide Assembler) ter še naketeri drugi, podpirajo Intelovo sintakso in se uporabljajo v okolju Windows. GAS (GNU Assembler) pa podpira obe vrsti sintakse, uporablja pa se večinoma v Linux okolju [5].

	Sintaksa	
Parameter	Intel	AT&T
Vrstni red parametrov	Ponor pred izvorom. Primer: "mov eax, 5" premakne v EAX register vrednost 5.	Izvor pred ponorom. Primer: "movl $5, eax".

Velikost parametrov	Velikost je izpeljana iz tipa registra, ki se uporablja.	Ukazu je dodana pripona, ki določa velikost parametrov. Pripone so naslednje: - 'q' označuje parameter velikosti 8 besed, - 'l' označuje parameter velikosti 4 besed, - 'w' označuje parameter velikosti 2 besed, - 'b' označuje parameter velikosti 1 besede.
Tip parametrov	Zbirnik sam prepozna, ali gre pri parametrih za register, konstante ali drug tip parametra.	Konstante imajo predpono $, za registre se uporablja predpona %.
Relativno naslavljanje	Označba za relativno naslavljanje so oglati oklepaji, prav tako pa je potrebno dodati predpono za velikost operanda. Primer: "mov eax, dword ptr [ebx + ecx*4 + lokacija]".	Za naslavljanje se uporabljajo okrogli oklepaji, zaporedje pa je baza, indeks ter obseg. Primer: "movl lokacija (%ebx, %ecx, 4), %eax".

Tabela 2.2: Razlike med Intel-ovo in AT&T sintakso

2.2.1 Načini izvajanja zbirnika x86

Procesorji, kompatibilni z x86 zbirnikom, podpirajo pet načinov, v katerih se lahko izvaja programska koda. Načini so naslednji [6]:

- realni način (angl. real mode),
- zaščitni način (angl. protected mode),

- 64-bitni način (angl. long mode),
- navidezni način (angl. virtual 86 mode),
- sistemsko-upravljalni način (angl. system management mode).

Realni način:
Realni način je podprt od serije 8086 procesorjev naprej, uporablja 20-bitni segmentirani naslovni prostor, kar omogoča naslavljanje 1MB pomnilniškega prostora. Realni način omogoča neposredni dostop do pomnilniškega prostora, vhodno-izhodnih naslovov ter zunanjih naprav. Ta način ne pozna večopravilnosti, prav tako ni zaščite pomnilnika ter priviligiranih kodnih obročev, ki bi ločevali uporabniško programsko kodo od sistemske. V tem načinu delujejo nekateri starejši operacijski sistemi (npr. MS-DOS) ter programska oprema kot je BIOS. Ta način uporabljajo vsi današnji procesorji ob zagonu, preden preklopijo v zaščitni ali kateri koli drug način.

Zaščitni način:
Zaščitni način se je prvič pojavil leta 1982 z izdajo procesorja serije 80286. Ta način podpira večopravilnost, koncept navideznega pomnilnika, ostranjevanje pomnilnika ter še nekatere druge posebnosti, ki dajo operacijskemu sistemu večji nadzor nad uporabniškimi aplikacijami.

64-bitni način:
64-bitni način uporabljajo 64-bitni operacijski sistemi in aplikacije, pojavil pa se je z AMD procesorji serije Athlon64 leta 2003. Omogoča dostop do 64-bitnih ukazov in registrov, nima pa še popolne podpore za 64-bitno naslavljanje pomnilniškega prostora. Ta način podpira zaščitni način, v katerem se izvajajo 16 in 32-bitni programi, ne podpira pa realnega in navideznega načina.

Navidezni način:
Navidezni način se je pojavil s serijo procesorjev 80386 leta 1985. Uporablja se za izvajanje aplikacij v Windows in OS/2 operacijskih sistemih, ki potrebujejo realni način za svoje izvajanje. Uporablja se isti način segmentacije, kot se uporablja v realnem načinu, vseeno pa ta način omogoča ostranjevanje ter zaščito pomnilniških strani kot v zaščitnem načinu.

Sistemsko-upravljalni način:

Sistemsko-upravljalni način je način, v katerem se ustavi izvajanje vse programske kode, tako uporabniških aplikacij kot samo izvajanje operacijskega sistema. Procesor začne izvajati poseben program (strojno programsko opremo ali strojno odvisen razhroščevalnik), ki ima neomejen dostop do pomnilnika, vhodno-izhodnih naprav ter notranjih virov strojne opreme. Ta način se je prvič pojavil s procesorjem i386SL leta 1995, uporablja pa se v naslednjih primerih:

- ob napakah strojne opreme,
- ob izvajanju funkcij, ki ščitijo strojno opremo (pred pregrevanjem procesorja itd.),
- ob posodobitvah strojne programske opreme,
- ta način za svoje izvajanje uporabljajo tudi nekateri zlonamerni programi, kot so korenski kompleti (angl. rootkits).

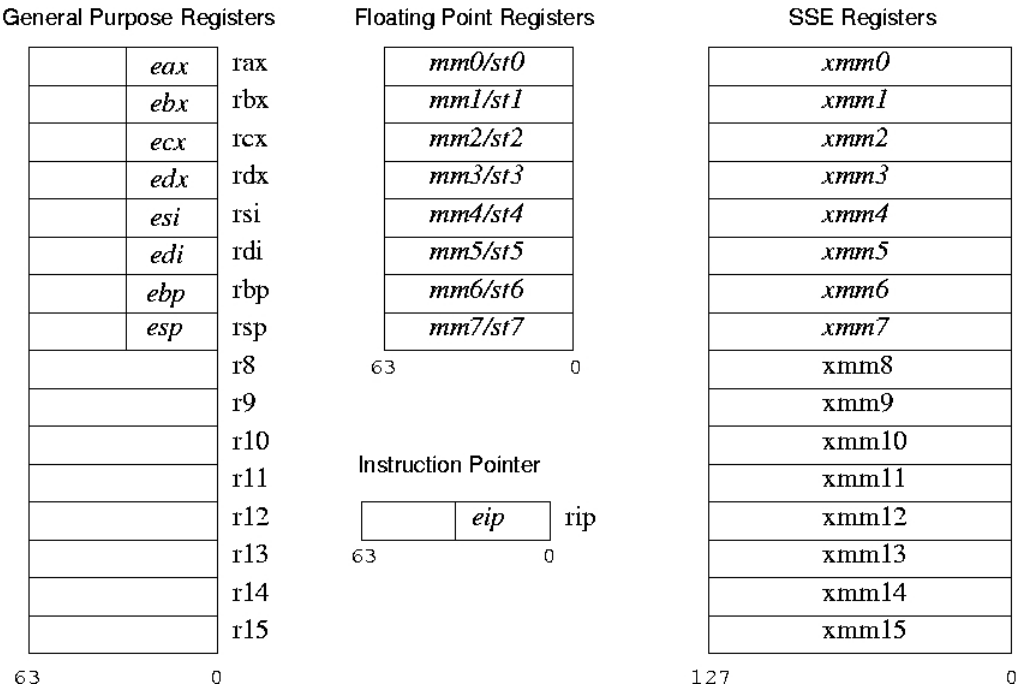
2.2.2 Registri

Ne glede na to, v kakšnem načinu se izvaja programska oprema na procesorju, za svoje izvajanje potrebuje procesorske registre. Procesorski register si lahko predstavljamo kot zelo hiter in majhen pomnilnik, vgrajen neposredno v samo procesorsko enoto. V procesorju je vgrajenih kar nekaj registrov, ki jih aplikacije lahko uporabljajo. Procesor ima splošno uporabne registre (slika 2.2), FPU registre (angl. flotaing point unit registers), MMX (angl. matrix math extension), SSE (angl. streaming simd extension) ter še nekatere druge. MMX registri se uporabljajo predvsem za paketno obdelavo podatkovnih tipov. To pomeni, da se ne uporabi cel register za operacijo nad 64-bitno vrednostjo, ampak se v register naloži dve 32-bitni, štiri 16-bitne ali osem 8-bitnih vrednosti, ki jih nato procesor hkrati obdela z enim ukazom. SSE registri so 128-bitni, večinoma pa se uporabljajo za grafično obdelavo. V diplomski nalogi smo uporabili splošno uporabne registre, kontrolne registre (angl. control registers), razhroščevalne registre (angl. debug registers) ter EFLAGS register.

Splošno uporabni registri:

Splošni registri se uporabljajo za različne operacije v procesorju, ločimo pa jih lahko glede na njihovo tipično uporabo v naslednje skupine:

- AX / EAX / RAX registri se imenujejo akomulatorji (angl. accumulator), uporabljajo se za splošne računske operacije, služijo pa tudi za shranjevanje vrednosti, ki jo vračajo funkcije.



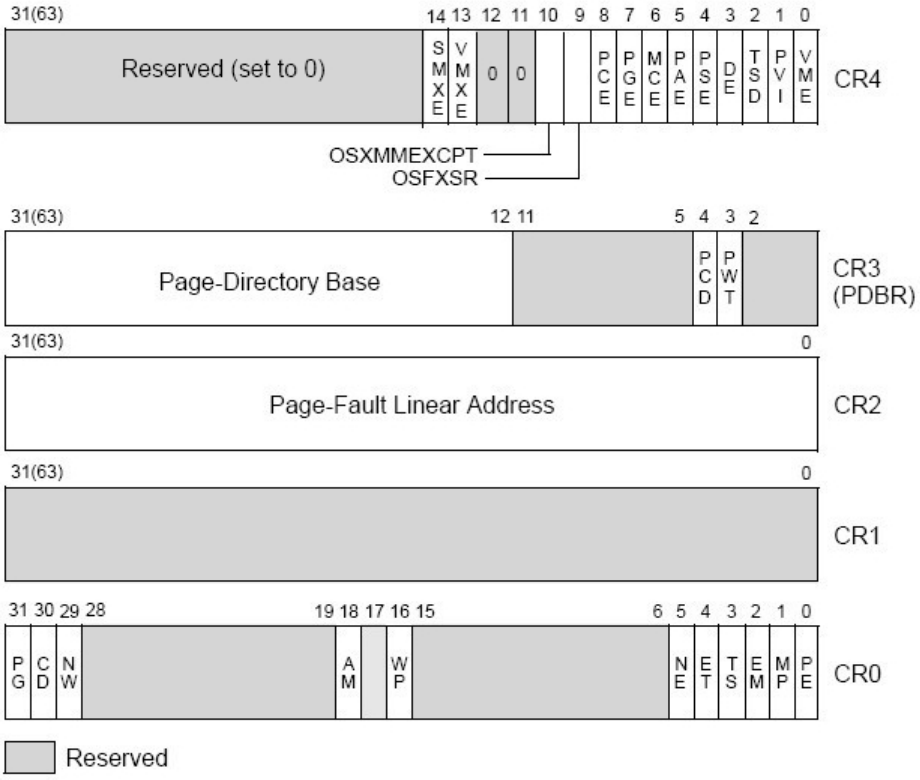
Slika 2.2: Splošno uporabni, FPU in SSE registri

- BX / EBX / RBX registri so bazni registri (angl base register), uporabljajo se največkrat za kazalce do tabel ali drugih struktur.
- CX / ECX / RCX se uporabljajo kot števci (angl. counter register), prav tako so nekateri zbirni ukazi odvisni od teh registrov (npr.: LOOP, LOOPE, LOOPNE itn.).
- DX / EDX / RDX se uporabljajo kot podatkovni registri (angl. data register).
- SI / ESI / RSI se uporabljajo kot izvorni registri (angl. source register) pri manipulacijah s pomnilnikom, največkrat pri operacijah z nizi.
- DI / EDI / RDI se uporabljajo kot ponorni registri (angl. destination register) pri operacijah z nizi in pomnilnikom.
- SP / ESP / RSP so kazalci na začetek sklada (angl. stack pointer).
- BP / EBP / RBP so bazni kazalci (angl. base pointer) na trenutni okvir sklada (angl. stack frame).
- IP / EIP / RIP so registri, ki vsebujejo kazalec na ukaz (angl. instruction pointer), ki se trenutno izvaja.

- registri R8 - R15 so splošni registri, ki pa so dostopni samo v 64-bitnih sistemih.

Kontrolni registri:

Kontrolni registri (slika 2.3) kontrolirajo samo izvajanje procesorja. Označeni so od CR0 do CR4, vsi pa so 32 ali 64-bitni, odvisno od zgradbe procesorja.



Slika 2.3: Kontrolni registri

CR0: register vsebuje informacije glede ostranjevanja, predpomnenja in še nekatere druge, ki so podrobneje opisane v tabeli 2.3.

Bit	Oznaka	Ime	Opis
31	PG	ostranjevanje (angl. paging)	Če je bit postavljen na 1, pomeni, da je ostranjevanje omogočeno, če ne je onemogočeno.

30	CD	predpomnenje (angl. caching)	1 omogoča predpomnenje, 0 ga onemogoča.
29	NW	predpomnenje z zapisovanjem nazaj (angl. write-back caching)	1 omogoča, 0 onemogoča write-back pomnenje.
28 - 19		Biti so rezervirani za prihodnjo uporabo.	
18	AM	poravnalna maska (angl. alignment mask)	1 pomeni, da je poravnavanje omogočeno, 0, da je onemogočeno.
17		Bit je rezerviran za prihodnjo uporabo.	
16	WP	zaščita proti pisanju (angl. write protect)	1 pomeni, da procesor lahko piše na strani označene samo za branje, 0, da ne more.
15 - 6		Biti so rezervirani za prihodnjo uporabo.	
5	NE	številska napaka (angl. numeric error)	1 pomeni, da je omogočeno interno lovljenje napak v FPU registrih, 0, da je onemogočeno.
4	ET	razširitveni tip (angl. extension type)	1 pomeni, da procesor podpira ukaze za matematični koprocesor, 0, da jih ne.
3	TS	zamenjava opravila (angl. task switched)	1 pomeni, da procesor omogoča zamenjavo opravila, 0, da ne omogoča.
2	EM	posnemanje (angl. emulation)	1 pomeni, da procesor ne podpira FPU ukazov, 0 pomeni, da procesor podpira FPU ukaze.
1	MP	nadzornik koprocesorja (angl. monitor coprocessor)	Bit nastavljen na 1 pomeni, da ukaza za zamenjavo opravila WAIT ali FWAIT preverita bit TS ter na podlagi tega generirata izjemo (če je nastavljen na 1). V nasprotnem primeru ukaza TS bita ne upoštevata.

0	PE	zaščitni način	1 pomeni, da je sistem v zaščitnem načinu, 0 pomeni, da je v realnem.

Tabela 2.3: Zgradba CR0 registra [7]

CR1: register je rezerviran za prihodnjo uporabo in se ga ne uporablja.

CR2: vsebuje linearni naslov strani, kjer se pojavi napaka ostranjevanja (angl. page fault linear address).

CR3: register se uporablja pri virtualnem naslavljanju (bit PG v CR0 registru nastavljen na 1). V tem primeru zgornjih 19 bitov (ali 51 bitov v 64-bitnih sistemih) predstavlja PDBR register (angl. page directory base register), ki se uporablja za pretvorbo virtualnega naslova v fizični. Register vsebuje še dva bita, PCD (angl. page level cache disable) in PWT (angl. page level write trough). Kombinacija bitov določa dostop do struktur v ostranjevalni arhitekturi (angl. paging structure hierarchy). Če je ostranjevanje onemogočeno, se vrednosti bitov ne upoštevata.

CR4: register se uporablja v zaščitnem načinu, vsebuje pa informacije o podpori navideznemu načinu, vhodno-izhodnih prekinitvah in še nekatere druge informacije. Pomen posameznih bitov je opisan v tabeli 2.4.

Bit	Oznaka	Ime	Opis
31 - 15		Biti so rezervirani za prihodnjo uporabo.	
14	SMXE	razširitev za SMX (angl. safe mode extension)	1 pomeni, da procesor podpira SMX razširitev, 0, da jo ne.
13	VMXE	razširitev za navidezni stroj (angl. virtual machine extension)	1 pomeni podporo za navidezni stroj, 0 pomeni, da podpore ni.
12 - 11		Biti so rezervirani za prihodnjo uporabo.	
10	OSXM-MEXCPT	podpora za nemaskirano FPE izjemo (angl. operating system support unmasked floating point exception)	1 pomeni, da procesor podpira prekinitev, 0, da jo ne.

9	OSFXSR	podpora za FXSAVE in FXRSTORE ukaza (angl. operating system support for FXSAVE and FXRSTORE)	1 pomeni, da sistem podpira ukaza, 0, da ne.
8	PCE	zmogljivostni števec omogočen (angl performance monitor counter enable)	1 pomeni, da se ukaz RDPMC lahko izvaja, v katerem koli priviligiranem obroču, 0 pomeni, da se ukaz lahko izvaja le v obroču 0.
7	PGE	globalno ostranjevanje omogočeno (angl. global page enable)	1 pomeni, da je globalno ostranjevanje omogočeno, 0, da je onemogočeno.
6	MCE	strojno preverjanje omogočeno (angl. machine check enable)	1 pomeni, da je strojno prekinitveno preverjanje omogočeno, 0 da je onemogočeno.
5	PAE	razširitev fizičnega naslova (angl. physical address extension)	1 omogoča razširitev fizičnega naslova čez 32 bitov, 0 onemogoča razširitev.
4	PSE	razširitev velikosti strani (angl. page size extension)	1 omogoča 32-bitno ostranjevanje, 0 ga onemogoča.
3	DE	razhroščevalna razširitev (angl. debugging extension)	Če je postavljen na 1, potem dostop do DR4 ali DR5 razhroščevalnih registrov generira izjemo. Če ima vrednost 0, sta DR4 in DR5 preslikavi registrov DR6 in DR7 (slika 2.4).
2	TSD	onemogočen časovni žig (angl. time stamp disabled)	1 pomeni, da je RDTSC ukaz onemogočen, 0 pomeni, da je omogočen.
1	PVI	navidezna prekinitev v zaščitnem načinu (angl. protected mode virtual interrupt)	1 pomeni, da so navidezne prekinitve omogočene, 0, da so onemogočene.

0	VME	razširitev navideznega načina (angl. virtual x86 mode extension)	1 pomeni, da so v načinu omogočene prekinitve in lovljenje izjem, 0, da niso.

Tabela 2.4: Zgradba CR4 registra [7]

Razhroščevalni registri:

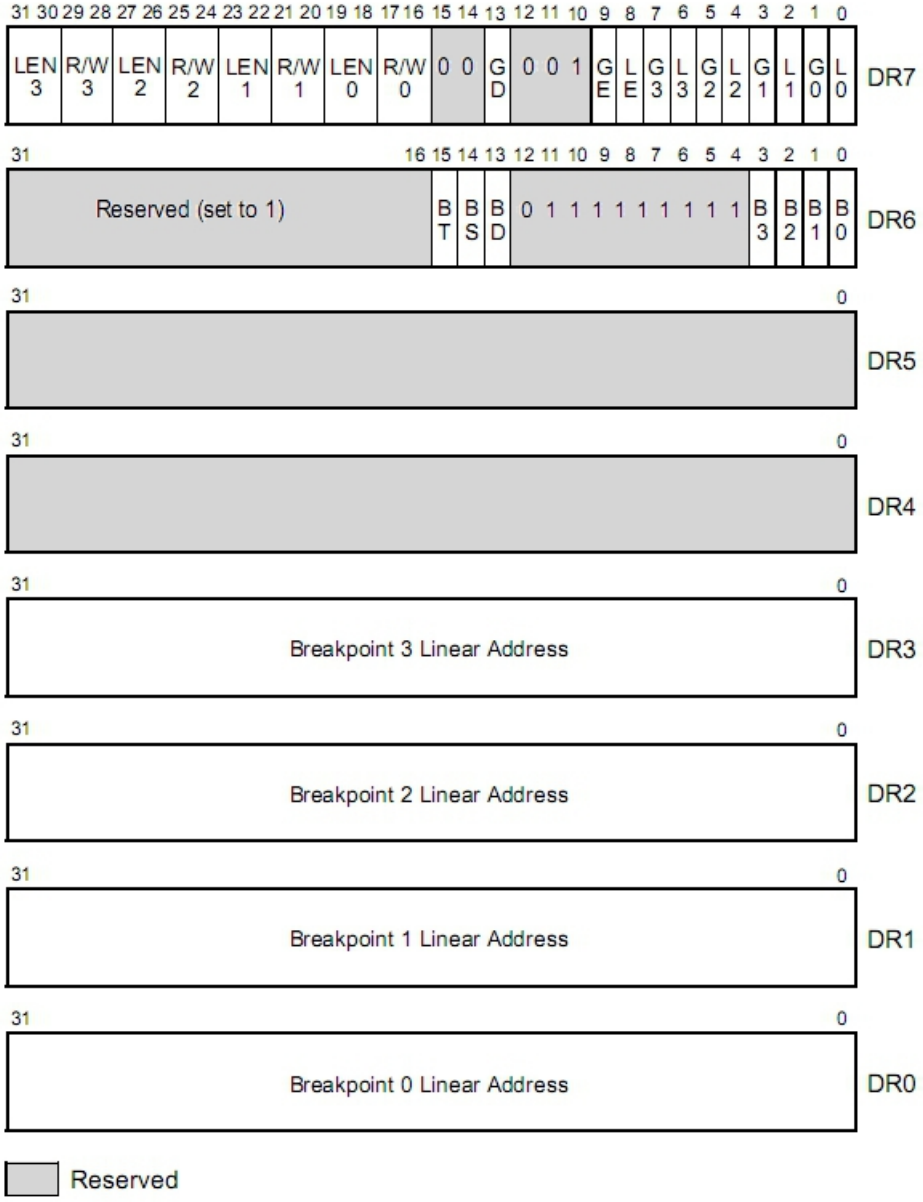
V procesorju je 8 razhroščevalnih registrov, označeni so od DR0 do DR7 (slika 2.4). Dostop do registrov je omejen, saj se do njih lahko dostopa le iz zaščitnega obroča 0. x86 arhitektura pozna štiri zaščitne obroče, označeni so s številkami od 0 do 3. Obroč 0 ima največ pravic in lahko dostopa do strojne opreme, v tem obroču se poganja jedro operacijskega sistema ter gonilniki. Obroč 3 ima najmanj pravic, v njem tečejo uporabniške aplikacije [8].Pomen posameznih registrov je naslednji:

DR0 - DR3: vsebujejo linearne naslove, kjer se nahajajo prekinitvene točke (angl. breakpoint conditions).

DR4 - DR5: če je bit DE v CR4 registru postavljen na 1 (tabela 2.4), potem sta registra nedostopna, dostop do njih povzroči izjemo. Če pa je bit postavljen na 0, potem je DR4 preslikava registra DR6, DR5 pa preslikava registra DR7.

DR6: register je statusni register in vsebuje informacije o prekinitvenih točkah ter še nekatere druge informacije. Opis posameznih bitov vsebuje tabela 2.5.

DR7: register je kontrolni register, vsebuje pa informacije o velikosti prekinitvenih točk, tipu, ter še nekatere druge informacije. Opis posameznih bitov se nahaja v tabeli 2.6.



Slika 2.4: Razhroščevalni registri

Bit	Oznaka	Opis
31 - 16		Biti so rezervirani za prihodnjo uporabo.
15	BT	Če je bit postavljen na 1, pomeni, da se je razhroščevalna izjema zgodila pri zamenjavi opravila.
14	BS	Bit postavljen na 1 pomeni, da se je razhroščevalna izjema zgodila v načinu korak za korakom (angl. single step execution mode). Ta način uporabljajo razhroščevalniki pri razhroščevanju programske opreme.
13	BD	1 pomeni, da naslednji strojni ukaz, ki ga bo procesor izvedel, dostopa do enega izmed razhroščevalnih registrov. Ta bit je omogočen, če je bit GD v DR7 registru postavljen na 1 (tabela 2.6).
12 - 4		Biti so rezervirani za prihodnjo uporabo.
3 - 0	BR3 - BR0	Posamezni bit postavljen na 1 označuje, na katerem naslovu je prišlo do prekinitve. BR3 označuje prekinitev na naslovu, na katerega kaže DR3 register, BR2, na katerega kaže DR2 register itd.

Tabela 2.5: Zgradba DR6 registra [7]

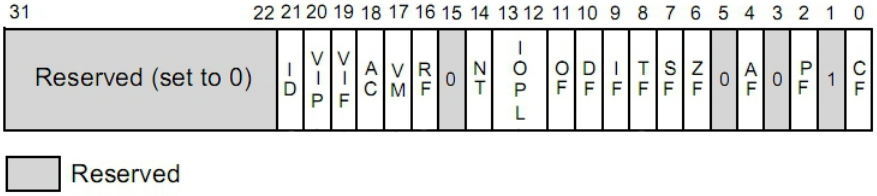
Bit	Oznaka	Opis
31, 30, 27, 26, 23, 22, 19, 18	LEN3 - LEN0	Kombinacija bitov (glej sliko 2.4) predstavlja velikost pomnilniške lokacije, kjer se nahaja prekinitvena točka. Kombinacije bitov označujejo naslednje velikosti: 00 - 1 beseda 01 - 2 besedi 10 - vrednost se ne uporablja 11 - 4 besede Posamezna velikost opisuje velikost prekinitvene točke na naslovu, na katerega kažejo registri od DR0 do DR3. Tako LEN3 določa velikost prekinitvene točke DR3 registra, LEN2 je za DR2 itd.

29, 28, 25, 24, 21, 20, 17, 16	RW3 - RW0	Kombinacija bitov določa tip prekinitvene točke. Če je bit DE v CR4 registru (tabela 2.4) postavljen na 1, potem kombinacija bitov določa naslednje tipe prekinitve: 00 - zgodi se ob izvedbi ukaza 01 - zgodi se ob zapisu podatka 10 - zgodi se ob vhodno-izhodni akciji 11 - zgodi se ob branju in zapisu podatkov Če je DE bit nastavljen na 0, potem je kombinacija bitov 10 neveljavna. Kombinacija RW3 določa tip prekinitvene točke v DR3 regitru, RW2 v DR2 registru itn.
15 - 14		Bita sta rezervirana za prihodnjo uporabo.
13	GD	Bit postavljen na 1 omogoča izvajanje prekinitve ob dostopu do katerega od razhroščevalnih registrov.
12 - 10		Biti so rezervirani za prihodnjo uporabo.
9, 8	GE, LE	Določata točno globalno in lokalno prekinitveno točko, bita se ne uporabljata in sta nastavljena na 1.
7, 5, 3, 1	G3 - G0	Biti določajo globalno prekinitveno točko. Taka prekinitev velja za vsa opravila. G3 določa globalno prekinitev za prekinitveno točko v DR3 registru, G2 v DR2 registru itd.
6, 4, 2, 0	L3 - L0	Biti določajo lokalno prekinitveno točko. Prekinitev velja le za trenutno opravilo, ki se izvaja. L3 določa lokalno prekinitev za točko v DR3 registru, L2 v DR2 registru itd.

Tabela 2.6: Zgradba DR7 registra [7]

EFLAGS register:

EFLAGS register (slika 2.5) je statusni register v procesorju, ki vsebuje vrednosti, kot recimo: ali je rezultat operacije 0, je prišlo do prekoračitve, je rezultat pozitiven ali negativen ter še nekaj drugih vrednosti. Zgradba registra predstavlja tabela 2.7.



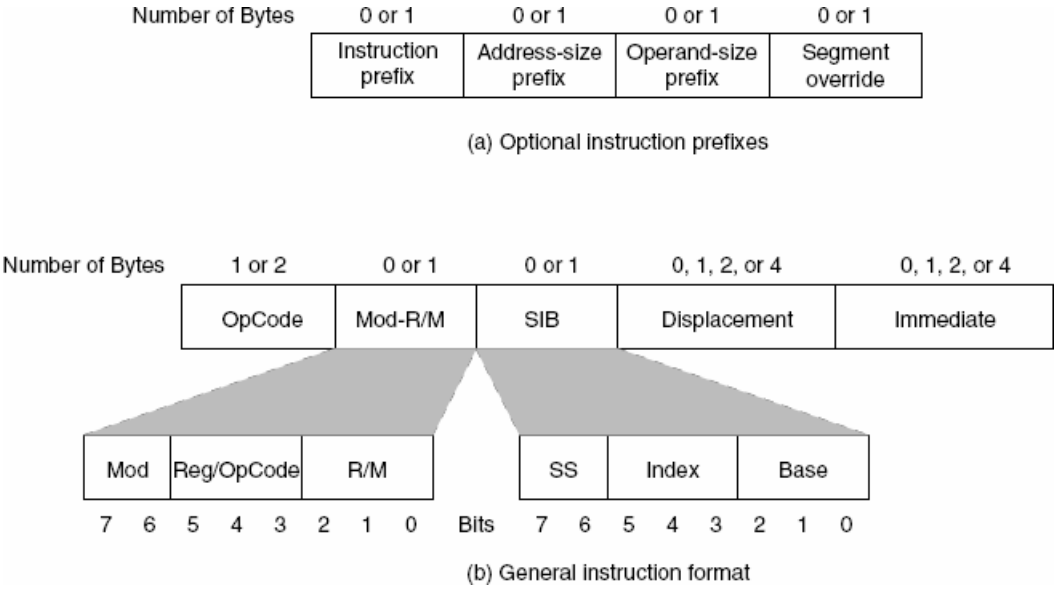
Slika 2.5: EFLAGS register

Biti	Oznaka	Opis
0	CF	Bit za preliv (angl. carry flag)
1	1	Rezervirano
2	PF	Paritetni bit (angl. parity flag)
3	0	Rezervirano
4	AF	Uravnalni bit (angl. adjust flag)
5	0	Rezervirano
6	ZF	Ničelni bit (angl. zero flag)
7	SF	Predznačni bit (angl. sign flag)
8	TF	Razhroščevalni bit (angl. trap flag)
9	IF	Prekinitveni bit (angl. interrupt flag)
10	DF	Smerni bit (angl. direction flag)
11	OF	Prekoračitveni bit (angl. overflow flag)
12 - 13	IOPL	Vhodno-izhodni privilegirani nivoji (angl. I/O privilege level)
14	NT	Vgnezdeno operacijski bit (angl. nested task flag)
15	0	Rezervirano
16	RF	Povrnitveni bit (angl. resume flag)
17	VM	Bit za navidezni način (angl. virtual 8086 mode flag)
18	AC	Bit za preverjanje poravnave (angl. alignment check flag)
19	VIF	Navidezni prekinitveni bit (angl. virtual interrupt flag)
20	VIP	Navidezni prekinitveno-čakalni bit (angl. virtual interrupt pending flag)
21	ID	Bit določa ali procesor podpira CPUID ukaz
22 - 31	0	Rezervirano

Tabela 2.7: Zgradba EFLAGS registra

2.2.3 Format zbirnega jezika

Format zbirnega jezika x86 sodi pod CISC arhitekturo (angl. complex instruction set computer), kar pomeni, da vsebuje veliko število različno velikih ukazov za razliko od RISC arhitekture (angl. reduced instruction set computer), kjer so vsi ukazi tudi enake dolžine. Tako je lahko dolžina zbirnega ukaza od 1 do 15 besed, kar oteži samo dekodiranje ukaza, saj je potrebno za vsak ukaz posebej izračunati njegovo dolžino. Format ukaza prikazuje slika 2.6.



Slika 2.6: Format x86 zbirnega ukaza

Bistveni sestavni deli ukaza so:

- Predpone ukaza (angl. instruction prefixes) so velikosti 1 besede, vsebujejo pa ukaze navedene v tabeli 2.8.

- Predpone segmenta (angl. segment override prefixes) so velikosti 1 besede, določajo pa, kateri segmentni register se bo uporabil v ukazu, ali pa kakšne velikosti je sam ukaz (16 ali 32-bitni). Predpone so opisane v tabeli 2.9.

Predpona	Operacijska koda	Opis
REP, REPE/REPZ	0xF3	Predpona se uporablja pri operacijah z nizi. Izvaja se, dokler je ZF zastavica v EFLAGS registru postavljena na 1.
REPNE/ REPNZ	0xF2	Predpona ima obraten pomen kot predhodna.
LOCK	0xF0	Izvedba ukaza s to predpono je atomska operacija.

Tabela 2.8: Predpone x86 zbirnega ukaza

Register	Operacijska koda	Opis
ES	0x26	Ekstra segment (angl. extra segment) se uporablja v ukazih, ki operirajo nad nizi.
CS	0x2E	Kodni segment (angl. code segment), ki ga vsebuje CS register, se uporablja za pridobivanje procesorskih ukazov, prav tako vsebuje informacije o privilegiranem nivoju, v katerem se bo ukaz izvedel (spodnja 2 bita določata nivo).
SS	0x36	Segment sklada (angl. stack segment) se uporablja pri ukazih, ki operirajo s skladom (PUSH, POP itd.).
DS	0x3E	Podatkovni segment (angl. data segment) se uporablja za dostope do podatkov v podatkovni sekciji.
FS	0x64	F segmentni register kaže na strukturo (vsaka nit ima svojo), ki vsebuje informacije o izjemah ter še nekatere informacije, ki so vezane na posamezno nit.
	0x66	Predpona se uporablja za razločevanje med 16 in 32-bitnim operandi v ukazu.
	0x67	Predpona se uporablja za razločevanje med 16 in 32-bitnim naslavljanjem v ukazu.

Tabela 2.9: Predpone segmenta

- Operacijska koda (angl. opcode) je lahko velikosti 1 ali 2 besedi, nekateri ukazi pa imajo tudi tretjo besedo, ki je vkodirana v "MOD-RM" polje. 2-besedna koda uporablja predpono, tako je prva beseda vedno 0x0F, druga beseda pa predstavlja dejanski ukaz. Prvega 2 bita operacijske kode se obravnavata kot zastavici, ki imata naslednji pomen:

  - Prvi bit je označen kot "s" in če je postavljen na 0, pomeni, da so operandi ukaza 8-bitni, če je na 1, pa pomeni, da so 16 ali 32-bitni.
  - Drugi bit se označuje kot "d" in predstavlja, kateri operand je izvorni in kateri ponorni.

  Če vzamemo za primer ukaz "mov eax,ebx", ki v EAX register premakne vrednost EBX registra, vidimo, da je njegova operacijska koda 0x89, kar pomeni, da ima "s" bit postavljen na 1 (uporablja se 16 ali 32-bitne operande), "d" bit na 0 (EAX je ponorni register, EBX je izvorni). Tabela 2.10 prikazuje spreminjanje MOV ukaza glede na kombinacijo bitov "s" in "d".

Op. koda	MOD-RM beseda	'd' bit	's' bit	Zbirni ukaz
0x88	0xD8	0	0	mov al, bl
0x89	0xD8	0	1	mov eax, ebx
0x8A	0xD8	1	0	mov bl, al
0x8B	0xD8	1	1	mov ebx,eax

Tabela 2.10: Sintaksa zbirnega ukaza MOV glede na vrednost s in d bita

- MOD-RM beseda je sestavljena iz treh polj: "MOD", "REG" in "R/M" polja. "MOD" polje ima 2 bita, označuje pa naslavljanje, ki ga ukaz uporablja. Vrednosti so lahko naslednje:

  - 00 pomeni registersko indirektno naslavljanje, naslavljanje z uporabo SIB besede brez odmika (ko ima R/M polje vrednost 100), ali naslavljanje z odmikom (ko ima R/M polje vrednost 101),
  - 01 pomeni naslavljanje s predznačenim enobesednim odmikom, ki sledi,

  - 10 pomeni naslavljanje s predznačenim 4-besednim odmikom, ki sledi,
  - 11 pomeni naslavljanje registrov. V bistvu ne gre za naslavljanje, ampak ta dva bita povesta, da sta izvorni in ponorni operand registra.

"REG" polje vsebuje vrednost, ki predstavlja ponorni register v ukazu. Vrednosti so opisane v tabeli 2.11. "R/M" polje se uporablja v povezavi z MOD poljem, pove pa, kateri register se uporablja pri naslavljanju. Vrednosti si lahko ogledamo kar v tabeli 2.11, če "REG" vrednost nadomestimo z "R/M" vrednostjo. Obstaja nekaj kombinacij "R/M" in "MOD" polj, ki so izjeme in so opisane v tabeli 2.12.

REG vrednost	Ponorni register (32/16/8-bitni)
000	EAX / AX / AL
001	ECX / CX / CL
010	EDX / DX / DL
011	EBX / BX / BL
100	ESP / SP / AH
101	EBP / BP / CH
110	EDI / DI / DH
111	ESI / SI / BH

Tabela 2.11: Prikaz ponornih registrov glede na vrednost REG polja

MOD	R/M	Naslavljanje
00	100	Uporablja se SIB besedo.
01	100	SIB beseda ter 8-bitni odmik.
10	100	SIB beseda ter 32-bitni odmik.
00	101	Uporablja se samo 32-bitni odmik.

Tabela 2.12: Izjeme kombinacij polj R/M in MOD

- SIB beseda je sestavljena iz treh polj: skale, indeksa in baze. Skala vsebuje vrednost, ki je pomnožena z indeksom pri naslavljanju. Vrednosti so naslednje:

- 00 pomeni indeks pomnožen z 1,
- 01 pomeni indeks pomnožen z 2,
- 10 pomeni indeks pomnožen s 4,
- 11 pomeni indeks pomnožen z 8.

Polje indeks vsebuje podatke o registru, ki se uporablja kot indeks. Tip indeksnega registra si lahko pogledamo v tabeli 2.11, omeniti pa je treba, da se vrednosti "100" ne uporablja.

Polje baza vsebuje vrednost, ki določa, kateri register se uporablja kot bazni pri naslavljanju. Tip registra se lahko razbere iz tabele 2.11, omeniti pa velja neveljavno vrednost '101', ki se ne uporablja.

V primeru, da se uporablja neveljavne vrednosti za bazo in indeks, se ukaz obravnava kot neveljaven.

2.3 Vprašanje legalnosti obratnega inženiringa

Programska oprema spada med intelektualno lastnino in je zaščitena z zakonom o avtorskih pravicah, pa naj gre za odprtokodne ali zaprtokodne rešitve. Zakon ščiti izvedbo, ne pa tudi same ideje, kar pomeni, da s tem ne zavira samega tehničnega razvoja, temveč le preprečuje, da bi drugi kopirali programsko kodo ter jo posredovali kot svojo.

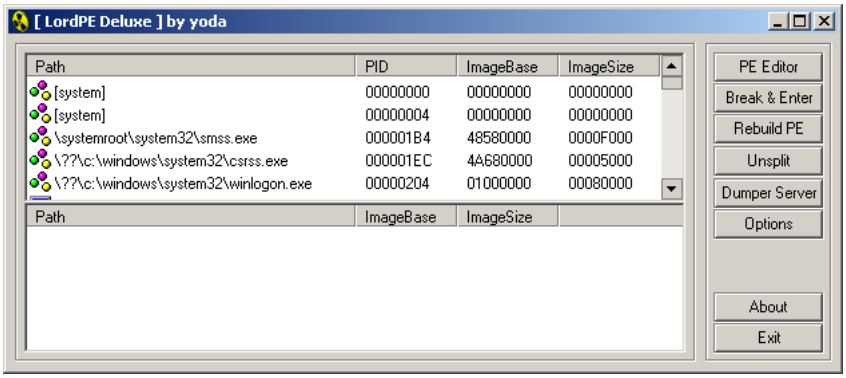
V Evropski Uniji to področje ureja direktiva 2009/24/EC [9], ki je razdeljena na 12 sklepov. Peti in šesti sklep se nanašata na obratni inženiring. V petem sklepu je navedeno, da je osebi, ki ima pravico uporabljati programsko opremo, dovoljeno brez privolitve avtorja analizirati, testirati ter opazovati programsko funkcionalnost, v kolikor to dela z namenom ugotavljanja same ideje, na kateri je napisana programska oprema. Šesti sklep pa dovoljuje, da se pridobljeno znanje in ideje uporabi v drugem programu z namenom zagovaljanja recipročnosti, ne sme pa se uporabiti programske kode, ki se je pridobila pri sami analizi. Dober primer sta projekta Samba ter WINE. Samba je implementacija Windows SMB/CIFS omrežnega protokola v operacijskem sistemu Linux. Protokol omogoča podporo za datotečne in tiskalniške servise za različne Windows odjemalce. WINE pa omogoča poganjanje Windows aplikacij na Linux okolju. Obratni inženiring je tudi omogočil kompatibilnost Microsoftovega Office datotečnega formata z OpenOffice pisarniško programsko opremo. Tako lahko rečemo, da je obratni inženiring v Evropski Uniji legalna dejavnost, v kolikor se ga uporablja za namene, navedene v direktivi.

Tega pa ne moremo trditi za druge države po svetu, ki uporabljajo drugačne dogovore. Tako recimo države, ki uporabljajo EULA licenco (angl. end-user license agreement), kot so Združene države Amerike, pogosto prepovejo izvajanje obratnega inženiringa nad programi pod to licenco. V teh primerih je izvajanje obratnega inženiringa nelegalno početje, razen če se seveda proizvajalec ter oseba ali skupina, ki bo izvajala obratni inženiring, drugače ne dogovorita [9, 10].

Poglavje 3

Orodja za izvajanje obratnega inženiringa

Za uspešno izvajanje obratnega inženiringa potrebujemo ustrezna orodja. V osnovi lahko orodja razdelimo na orodja, ki omogočajo statično ali dinamično analizo programa. Razhroščevalniki ter sistemski monitorji spadajo med orodja za dinamično analizo, medtem ko povratni zbirniki (angl. disassemblers) spadajo med orodja za statično analizo. Za analizo PE formata se uporabljata tudi orodji LordPe[1] (slika 3.1) in PEId[2]. PEId poleg tega omogoča prepoznavanje več kot 600 različnih programskih kodirnikov (angl. PE crypter) ter pakirnikov (angl. PE packer).



Slika 3.1: Grafični vmesnik programa LordPE Deluxe

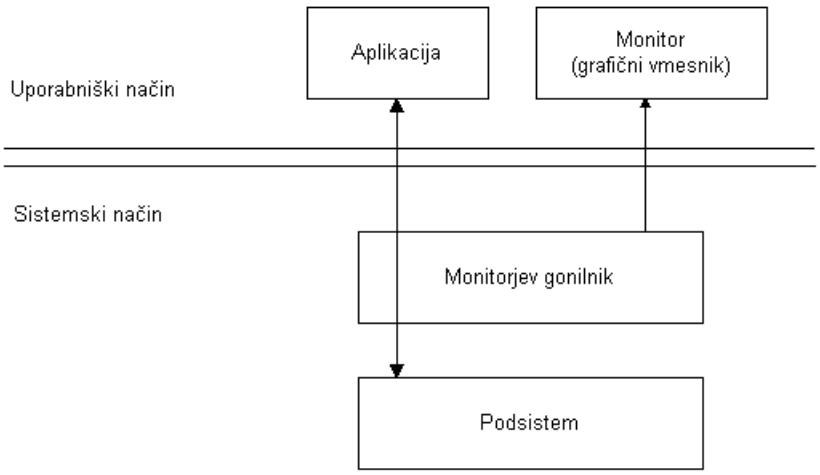
[1] http://www.woodmann.com/collaborative/tools/images/Bin_LordPE_2007-10-21_1.48_LordPE_1.41_Deluxe_b.zip

[2] http://www.peid.info/download.html

3.1 Sistemski monitorji

Sistemski monitorji spadajo med orodja za dinamično analizo programskih operacij. Uporabljajo se za nadzor operacij, ki jih program izvaja za branje določenih datotek ali dostop do registra. Taka programa sta Registry Monitor[3] in File Monitor[4], s katerima lahko nadzorujemo dostope do registra ter datotek na disku. Process Monitor vsebuje funkcije prej naštetih monitorjev, vključuje pa še podatke o procesih, mrežne aktivnosti procesa ter še mnogo drugih podatkov.

Večina zgoraj naštetih monitorjev deluje na podoben način (slika 3.2). Vsi vsebujejo gonilnik, ki se ob zagonu monitorja odpakira v sistemsko mapo, naloži se v pomnilnik, gonilnik pa se nato izbriše. Monitor nato s pomočjo gonilnika prestreza določene operacije (odvisno od tipa monitorja), ki jih izvajajo druge aplikacije ter jih zapisuje v dnevnik. Monitorji omogočajo nastavitev filtrov, kar nam olajša samo analiziranje operacij nad registrom ali datotečnim sistemom.

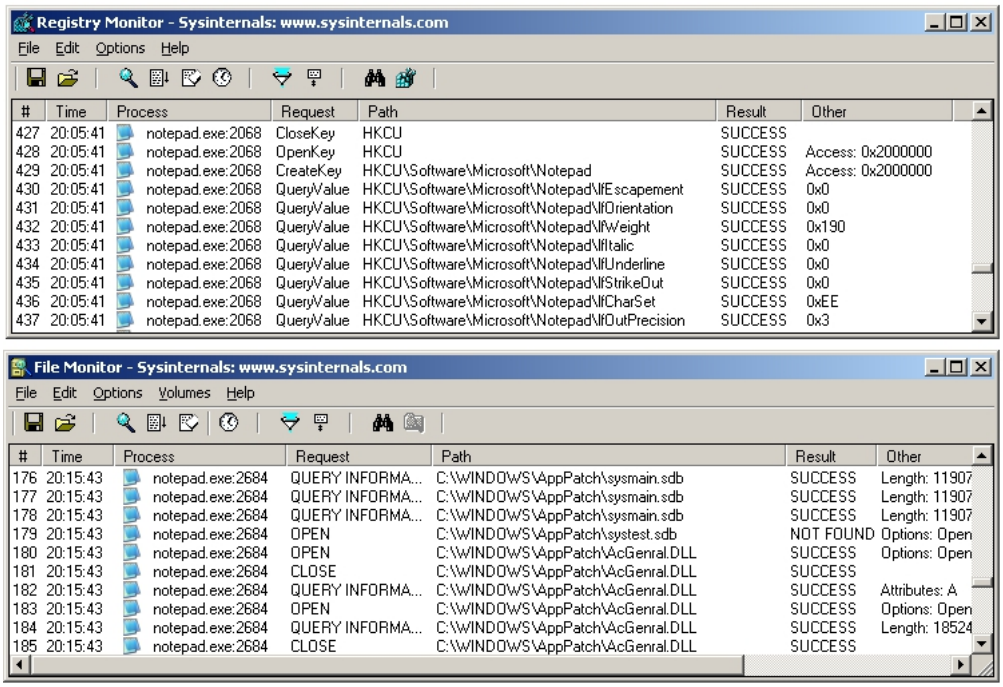


Slika 3.2: Delovanje sistemskega monitorja

[3] http://www.softpedia.com/get/Programming/Other-Programming-Files/Regmon.shtml

[4] http://www.softpedia.com/progDownload/Filemon-Download-3365.html

Grafična vmesnika orodij File Monitor in Registry Monitor sta popolnoma enaka (slika 3.3). Večji del vmesnika obsega prikazovalnik dogodkov, ki je razdeljen na sedem delov. Prva dva dela sta zaporedna številka dogodka ter čas, ko je bil dogodek zabeležen. Sledita mu ime procesa ter tip zahteve, ki je sprožila dogodek (odpiranje, branje, pisanje ipd.). Peti del vsebuje pot do datoteke ali ključa v registru, na katerega se zahteva nanaša. Zadnja dva dela sta rezultat zahteve in podrobnejši opis posameznih parametrov dogodka. Tako imamo pri dogodku, ko program dostopa do datoteke, napisano njeno velikost in atribute, pri dostopu do registrskega ključa pa vrednost le-tega.



Slika 3.3: Grafični vmesnik programov File Monitor in Registry Monitor

3.2 Razhroščevalniki

Razhroščevalniki omogočajo izvajanje strojne kode ukaz za ukazom ter nastavljanje prekinitvenih točk (angl. breakpoints), s tem pa nadzor nad izvajanjem samega programa. Razdelimo jih lahko na uporabniške ter sistemske razhroščevalnike. V uporabniškem načinu omogočajo razhroščevanje aplikacij,

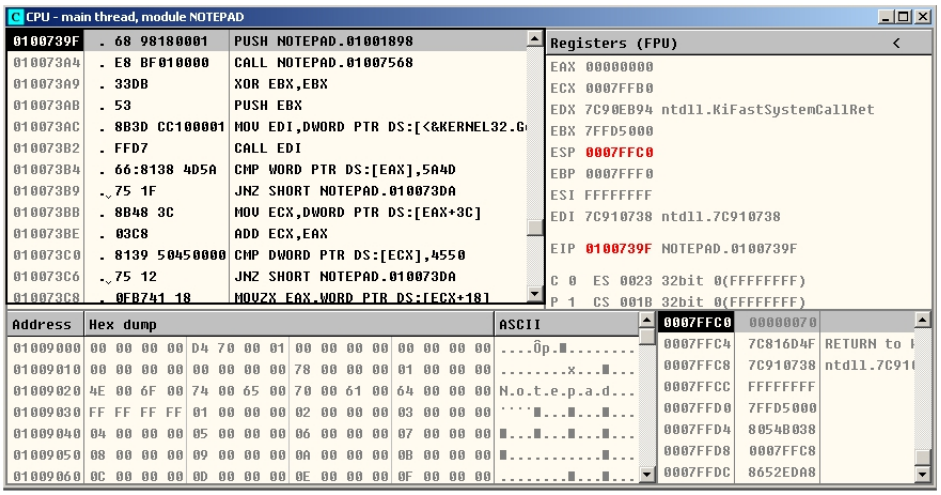
ne omogočajo pa razhroščevanja sistemskih gonilnikov. To omogočajo razhroščevalniki, ki delujejo v sistemskem načinu.

3.2.1 Razhroščevalniki v uporabniškem načinu

Razhroščevalniki v uporabniškem načinu izvajajo kodo, napisano za uporabniški način. Najbolj znani razhroščevalniki za obratni inženiring so OllyDbg[5], WinDbg[6] ter Immunity Debugger[7].

OllyDbg:

Razhroščevalnik (slika 3.4) vsebuje napreden obratni zbirnik, s katerim najprej analizira programsko kodo. Obratni zbirnik prepozna zanke, parametre funkcij ter nekatere podatkovne strukture. Omogoča vpogled v PE format aplikacije, vsebuje listo uvoznih funkcij ter še mnogo drugh podatkov. Vsebuje tudi podporo za vtičnike (angl. plugins), s katerimi lahko avtomatiziramo ter razširimo samo delovanje razhroščevalnika [11].

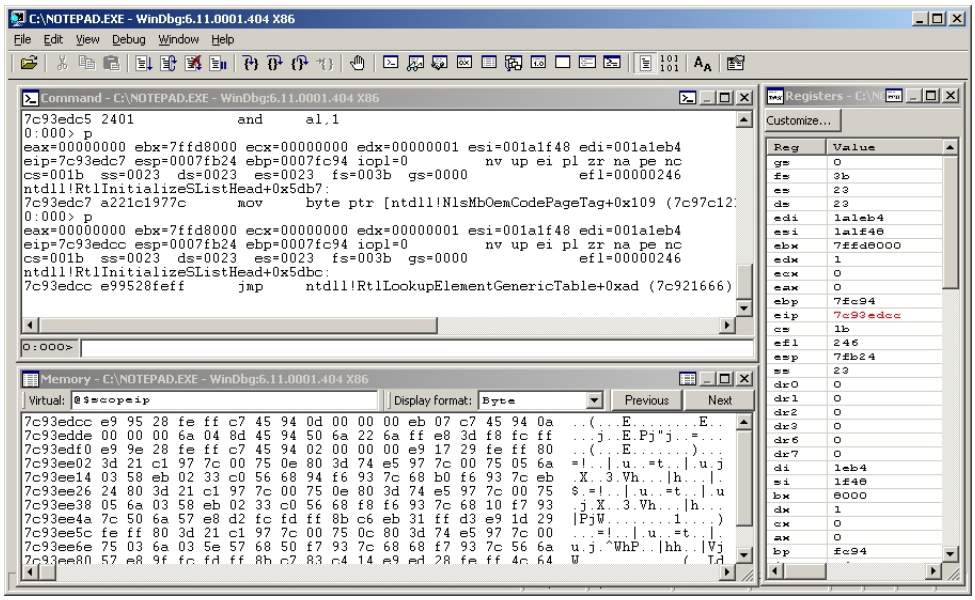


Slika 3.4: Grafični vmesnik programa OllyDbg

[5] http://www.ollydbg.de/download.htm
[6] http://www.windbg.org/
[7] http://debugger.immunityinc.com/register.html

WinDbg:

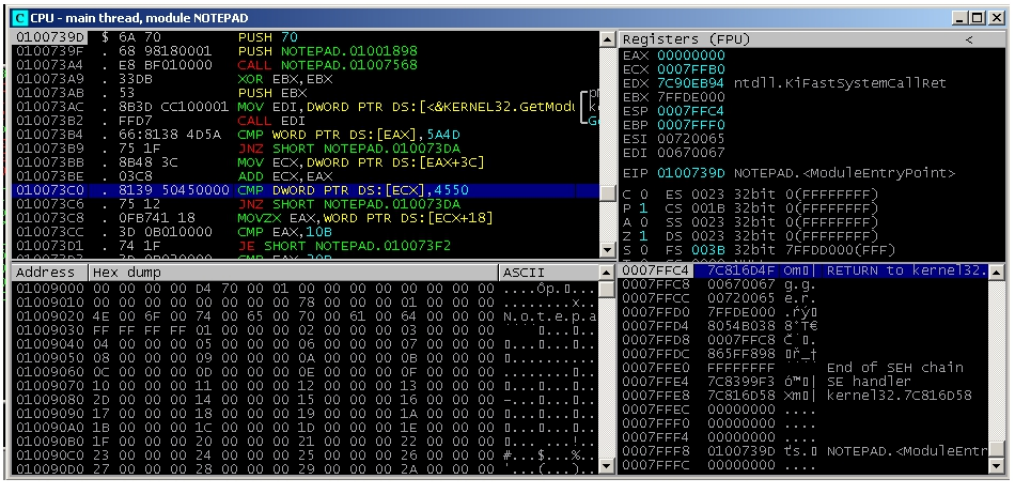
Glavna slabost razhroščevalnika (slika 3.5) je omejen obratni zbirnik, prednost pa ta, da omogoča shranjevanje kopice (angl. heap) tekočega procesa, PEB in TEB struktur ter še nekaterih podatkov. Tudi ta razhroščevalnik podpira vtičnike, kar omogoča razhroščevanje .NET aplikacij ter uporabo razhroščevalnika v navideznih strojih [12].



Slika 3.5: Grafični vmesnik programa WinDbg

Immunity Debugger:

Razhroščevalnik je posebej prilagojen analizi programske kode za prekoračitve kopice ali sklada, kar omogoča izvajanje izkoriščevalske kode (angl. exploit). Prav tako kot OllyDbg vsebuje napredni skriptni jezik, napredni povratni zbirnik za analizo zbirne kode, omogoča tudi povezavo s sorodnimi produkti, ki služijo za analizo izkoriščevalske kode. Omeniti še velja, da ima podporo za skriptni jezik Python, kar omogoča izvajanje skript neposredno v razhroščevalniku. Grafični vmesnik je zelo podoben vmesniku OllyDbg, kar je razvidno s slik 3.6 in 3.4 [13].



Slika 3.6: Grafični vmesnik programa Immunity Debugger

3.2.2 Razhroščevalniki v sistemskem načinu

Razhroščevalniki v sistemskem načinu omogočajo razhroščevanje gonilnikov ter jedra operacijskega sistema, njihov problem je v kompatibilnosti med operacijskimi sistemi, saj so precej odvisni od same zgradbe in delovanja le-teh. Znana razhroščevalnika sta Softice in HyperDbg[8].

Softice:

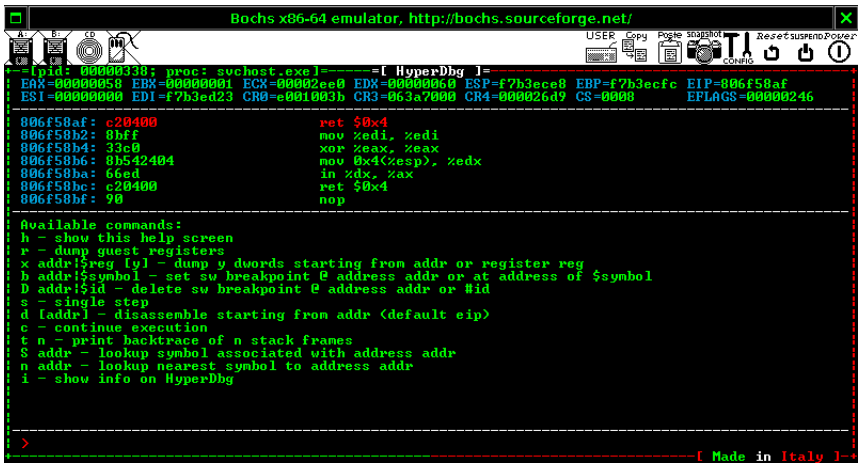
Razhroščevalnik (slika 3.7) je zasnovan tako, da se naloži pred operacijskim sistemom. Tako omogoča popoln nadzor nad sistemom ter aplikacijami, ki se na njem izvajajo. V začetku je bil napisan za operacijski sistem MS-DOS, kasneje so ga nadgradili s podporo za Windows. Aprila 2006 pa so prenehali z razvojem le-tega. Kot razlog so navedli tehnične težave pri zagotavljanju kompatibilnosti med različnimi verzijami operacijskega sistema Windows. Ima tudi podporo za razširitve, ki omogočajo vpeljavo dodatnih funkcionalnosti. Program je bil do leta 2007 dostopen na njihovi spletni strani[9] kot 14-dnevna preizkusna različica, ko so prenehali z zagotavljanjem podpore, pa so ga umaknili z uradnega spletnega mesta. Tako program lahko dobimo na neuradnih straneh[10], ki vsebujejo orodja za obratni inženiring [14].

[8] `http://code.google.com/p/hyperdbg/downloads/list`
[9] `http://www.compuware.com/`
[10] `http://cracklab.ru/download.php?action=get&n=NTc1`

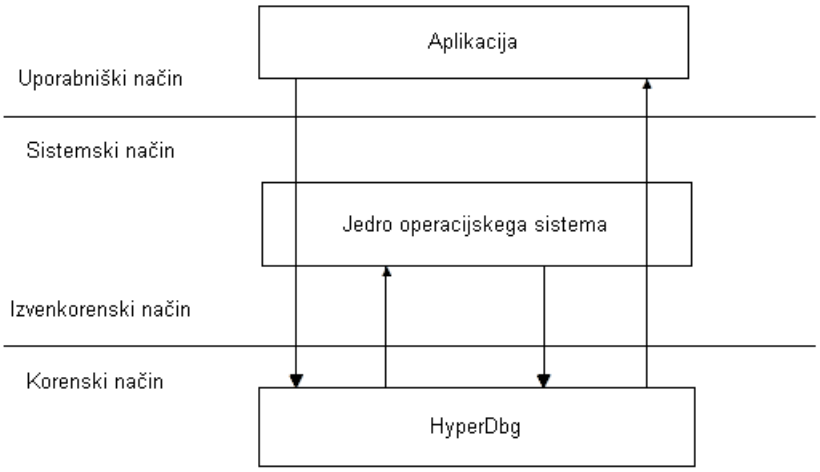
Slika 3.7: Grafični vmesnik programa Softice

HyperDbg:

Razhroščevalnik deluje kot majhen upravitelj navideznega stroja (angl. hypervisor), kar omogoča transparentno delovanje glede na operacijski sistem. Ravno ta način delovanja omogoča razhroščevanje tudi najbolj kritičnih točk sistema, kot so izvajanje prekinitev, izvajanje kode ob izjemah ter deljenje časovnih rezin v operacijskem sistemu. Razhroščevalnik je osnovan na Intelovi VT tehnologiji (angl. virtualization transformation), komunikacija s sistemom (slika 3.9) pa poteka preko gonilnika, ki se naloži ob zagonu razhroščevalnika. HyperDbg tako lahko uporabljamo, če imamo strojno podprto virtualizacijo, ali, če ga poganjamo v emulatorju (slika 3.8) [15].



Slika 3.8: Grafični vmesnik HyperDbg razhroščevalnika

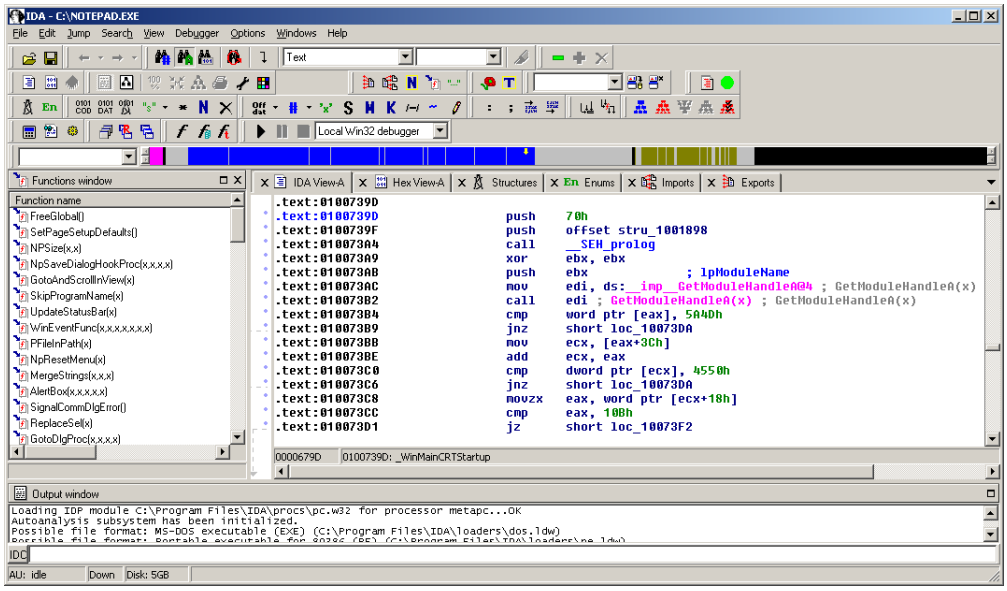


Slika 3.9: Delovanje HyperDbg razhroščevalnika

3.3 Obratni zbirniki

Obratni zbirniki spadajo med statična orodja, ki omogočajo pretvorbo strojne kode v zbirni jezik. V osnovi jih lahko ločimo glede na algoritem, ki ga uporabljajo za analizo kode. Tako imamo linearne in rekurzivne obratne zbirnike. Linearni zbirniki so hitri, njihova slabost pa je neupoštevanje dejanskega poteka izvedbe strojne kode. Rekurzivni upoštevajo tudi sam potek izvedbe, so pa zato počasnejši pri sami analizi. Eden boljših obratnih zbirnikov je IDA (Interactive Disassembler), ki podpira celo vrsto zbirnih jezikov, od x86, ARM, Motorola ter še mnogo drugih. IDA[11] vsebuje vgrajen razhroščevalnik, vendar se ga najpogosteje uporablja kot obratni zbirnik, saj omogoča prepoznavanje sistemskih struktur in API klicev. Uporaba modula FLIRT (angl. fast library identification and recognition tehnology) pa omogoča analizo in prepoznavanje statično povezanih knjižnic. IDA omogoča razširitve z vtičniki, omogoča pa tudi pisanje skript za avtomatizacijo operacij. Za analizo strojne kode uporablja tako linearni kot rekurzivni algoritem. IDA obratni zbirnik se dobi v plačljivi in neplačljivi različici. Slika 3.10 prikazuje neplačljivo različico, ki vsebuje podporo le za x86 zbirni jezik, medtem ko plačljiva vsebuje podporo tudi za ostale zbirne jezike, ki smo jih že prej omenili [1].

[11] `http://www.hex-rays.com/products/ida/index.shtml`

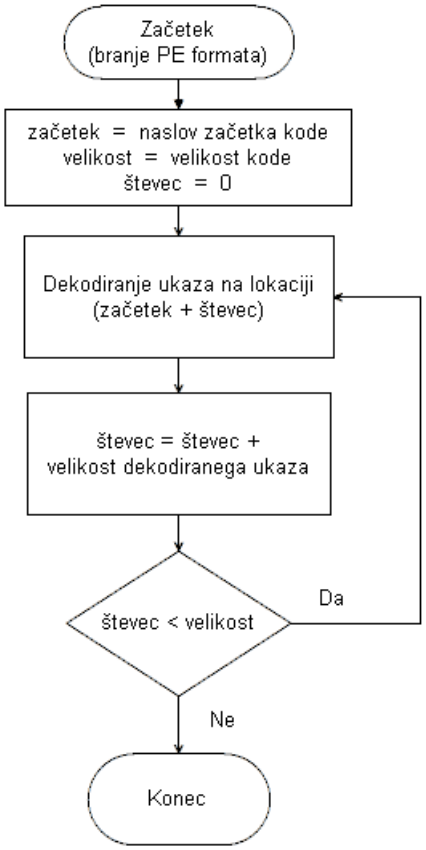


Slika 3.10: Grafični vmesnik IDA obratnega zbirnika

3.3.1 Linearni algoritem

Linearni algoritem (slika 3.11) je precej hiter pri analizi, njegova slabost je, da ne upošteva dejanskega poteka strojne kode. Algoritem začne analizo na začetku sekcije, ki vsebuje programsko kodo. Dekodira strojni ukaz, iz same strukture ukaza pa razbere njegovo dolžino. Dolžino prišteje naslovu, kjer je analiziral zadnji ukaz, tako dobi naslov naslednjega ukaza. V primeru, da se uporablja določene tehnike za obfuskacijo, tak algoritem ne bo pravilno analiziral kode, saj bo dejanski potek kode drugačen, kot pa ga trenutno razpozna algoritem.

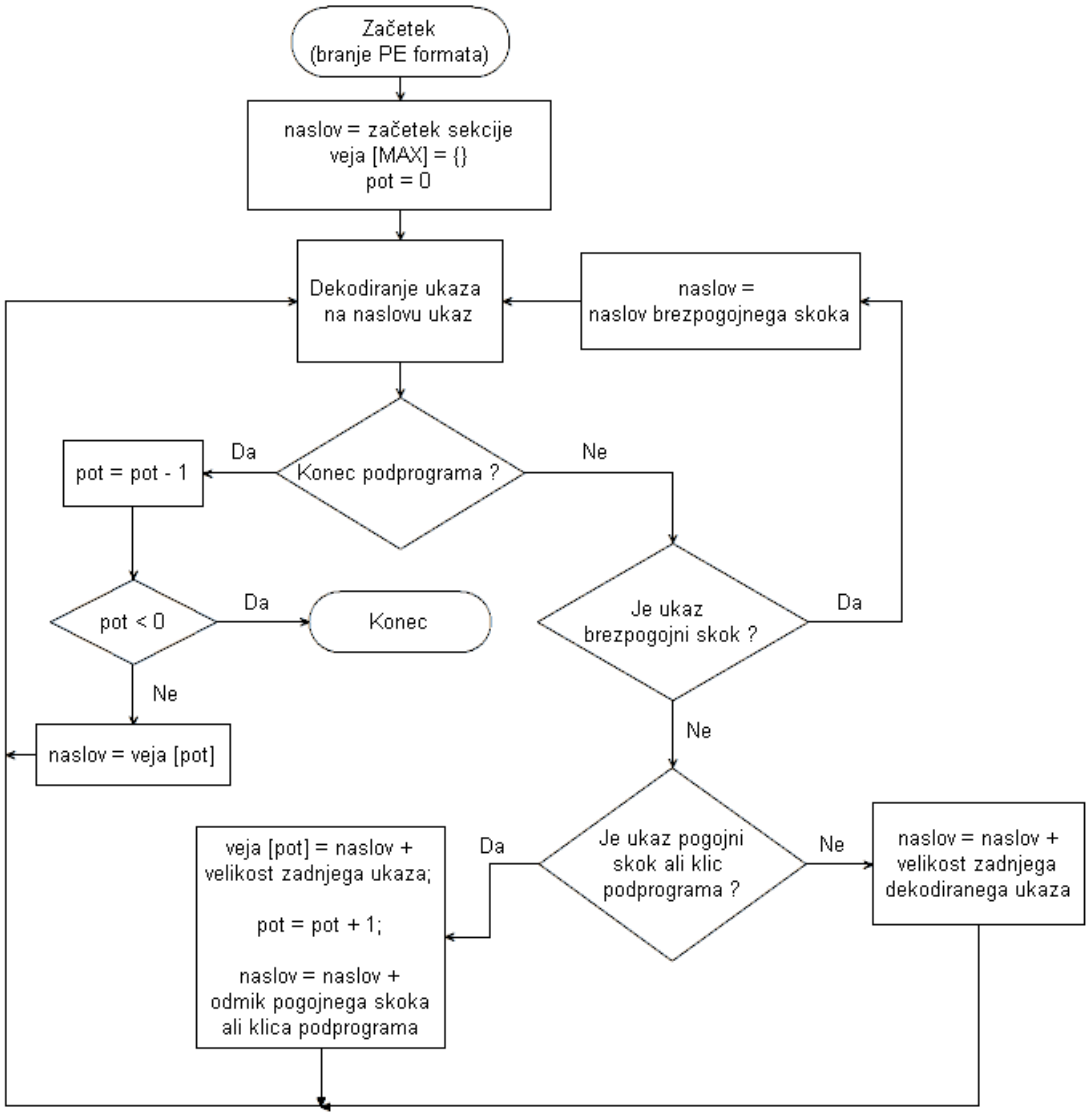
Obfuskacija je tehnika skrivanja programske kode, s katero otežimo samo analizo le-te. Obfuskacijo lahko izvajamo na dveh nivojih, na nivoju izvorne kode ali pa na strojnem nivoju. Izvorno kodo obfuskatorji spremenijo tako, da zamenjajo imena funkcij ter spremenljivk z določenim zaporedjem znakov. Taka obfuskacija ne vpliva na analizo strojne kode. Obfuskacija na strojnem nivoju je dodajanje odvečne kode med obstoječo, kar privede do napačne analize strojne kode. To tehniko smo podrobno opisali v naslednjem poglavju.



Slika 3.11: Diagram linearnega algoritma

3.3.2 Rekurzivni algoritem

Za razliko od linearnega, rekurzivni algoritem (slika 3.12) upošteva potek samega izvajanja strojne kode, zato je tudi uspešnejši pri sami analizi. Začetek je enak kot pri linearnem algoritmu, razlika je v tem, da si algoritem shrani še veje programa, ki nastanejo pri klicih procedure ali pri pogojnih skokih. Ko algoritem pride do ukaza, ki predstavlja klic procedure ali nepogojni skok, si shrani naslov, nato pa nadaljuje analizo na naslovu, na katerega kaže skok ali klic procedure.



Slika 3.12: Diagram rekurzivnega algoritma

Poglavje 4

Metode nasprotno-obratnega inženiringa

Obratni inženiring se ne uporablja samo za zagotavljanje recipročnosti ali za pridobivanje znanja, ampak tudi za odpravljanje raznih zaščit programske opreme. Med te zaščite spadajo zaščite proti izdelovanju nelegalnih kopij, algoritmi za licenciranje programskih kopij, razni DRM-ji (angl. digital rights managment) in podobno. Z uporabo obratnega inženiringa se nato te zaščite zaobide, kar omogoča nelegalno distribucijo in uporabo programske opreme. Programa se ne da popolnoma zaščititi pred modifikacijami, lahko pa se oteži sam proces obratnega inženiringa z uporabo različnih metod nasprotno-obratnega inženiringa (angl. anti-reverse engineering) [1, 16, 17]. Metode lahko razdelimo na:

- metode za zaznavanje monitorjev,
- metode za zaznavanje razhroščevalnikov,
- metode za onemogočanje analize obratnega zbirnika.

Za zaščito programske kode se uporabljajo tudi različni pakirniki, ki zapakirajo ter skrčijo program. Takega programa ni mogoče modificirati, saj se koda odpakira šele ob zagonu le-tega. Večina pakirnikov zapakira programsko kodo in spremeni PE format (izbriše uvozno tabelo), kar oteži samo uporabo razhroščevalnika ter obratnega zbirnika. Uporabljajo se tudi zgoraj naštete metode, da bi otežili izvajanje obratnega inženiringa nad pakirniki, saj bi to pomenilo, da lahko odpakiran program iz pomnilnika prenesemo na disk, popravimo PE format ter imamo tako delujoč odpakiran program. Takega lahko modificiramo ter zaobidemo morebitno zaščito.

4.1 Metode za zaznavanje monitorjev

Za zaznavanje monitorjev se uporabljajo metode, kot so iskanje gonilnikov, ki se naložijo ob zagonu, iskanje po imenu okna ali registriranem razredu, ki ga uporablja monitor za prikazovanje dogodkov. Lahko se uporabi tudi preverjanje imen procesov, ki se izvajajo na sistemu, vendar se to preprosto zaobide s spremembo imena datoteke.

4.1.1 Zaznavanje z iskanjem gonilnikov

S to metodo preverimo, če je naložen monitorjev gonilnik. Uporabimo Windows API klic CreateFile, ki nam vrne odprto ročico gonilnika. V primeru, da gonilnik ni naložen, ali da je prišlo do napake pri pridobivanju ročice do njega, nam vrne vrednost -1. Primer izvorne kode prikazuje izpis 4.1.

```
HANDLE handle = CreateFileA ("\\\\.\\FILEMON701",
                             (GENERIC_READ | GENERIC_WRITE),
                             (FILE_SHARE_READ | FILE_SHARE_WRITE),
                             0,
                             OPEN_EXISTING,
                             FILE_ATTRIBUTE_NORMAL,
                             0
                            );

DWORD lastError = GetLastError();

if (INVALID_HANDLE_VALUE == handle){
    if ((lastError == 4) || (lastError == 5)){
            MessageBoxA(0,"Gonilnik zaznan (NI DOSTOPA)","",0);
    }
    else{
        MessageBoxA(0,"Gonilnik ni zaznan","",0);
    }
}
else{
    CloseHandle(handle);
    MessageBoxA(0,"Gonilnik zaznan","",0);
}
```

Izpis 4.1: Programska koda za zaznavanje gonilnika Filemon 7.04 monitorja

Metoda zazna gonilnik tudi, če monitor ni pognan. Večina monitorjev namreč ob prenehanju delovanja ohrani gonilnik naložen zaradi stabilnosti operacijskega sistema. Slabost metode je ta, da je za zaznavanje gonilnika potrebno vedeti ime samega gonilnika, ki ga monitor uporablja. To se med verzijami lahko razlikuje, kar oteži samo zaznavanje le-tega.

4.1.2 Zaznavanje z iskanjem registriranega razreda

Okno, ki ga monitor uporablja za prikazovanje dogodkov, ima registriran razred, ki se registrira z Windows API funkcijama RegisterClass ali RegisterClassEx. Ta vrne atom razreda (angl. atom class), ki unikatno predstavlja registriran razred.

S to metodo (izpis 4.2) skušamo ugotoviti, če je določen razred registriran. Če je, potem se monitor izvaja. Za iskanje uporabimo Windows API klic FindWindowA, ki mu kot parameter podamo ime okna, registriran razred ali oboje. V primeru, da se v monitorju modificira ime razreda, ga metoda ne bo zaznala, kar je njena slabost.

```
HANDLE handle = FindWindowA("PROCMON_WINDOW_CLASS",0);

if(0 != handle){
  SendMessageA(handle,WM_SYSCOMMAND,(WPARAM)SC_CLOSE,0);
  if(0 != GetLastError()){
     MessageBoxA(0,"Napaka pri pošiljanju SC_CLOSE sporočila","",0);
  }
}
else{
  MessageBoxA(0,"Monitor ni zaznan","",0);
}
```

Izpis 4.2: Programska koda za onemogočanje programa Process Monitor 2.95

4.2 Metode za zaznavanje razhroščevalnikov

Obstajajo različni tipi razhroščevalnikov, ki pa imajo skupne lastnosti. Tako vsi uporabljajo prekinitvene točke za nadzor izvajanja programa, prav tako se ob uporabi razhroščevalnika nastavijo določene vrednosti v EFLAGS registru ter razhroščevalnih registrih, kar lahko uporabimo pri zaznavanju. V primeru,

da se izvaja program v razhroščevalniku korak za korakom (angl. single step), pride pri izvajanju same kode do večjih časovnih zamikov, kot če se program izvaja brez razhroščevalnika. Tako lahko z merjenjem časovnih zamikov pri izvajanju programske kode zaznamo, ali se program izvaja v razhroščevalniku. Za zaznavanje se uporabi Windows API klice (npr. IsDebuggerPresent, DebugBreak ...), ki jih vsebuje operacijski sistem.

4.2.1 Zaznavanje s pomočjo API klicev

Windows vsebuje kar nekaj API klicev, s katerimi se da zaznati razhroščevalnik. Uporabi se lahko naslednje klice:

- IsDebuggerPresent,
- NtQueryInformationProcess,
- DebugBreak.

IsDebuggerPresent:
Izpis 4.3 prikazuje uporaba API klic IsDebuggerPresent. Klic vrne vrednost 0, če se trenutni proces ne izvaja v uporabniškem razhroščevalniku. Če se, vrne vrednost različno od 0.

```
if(0 == IsDebuggerPresent()){
    MessageBoxA(0,"Razhroščevalnik ni zaznan","",0);
}
else{
    MessageBoxA(0,"Razhroščevalnik zaznan","",0);
}
```

Izpis 4.3: Primer uporabe IsDebuggerPresent API klica za zaznavanje razhroščevalnika

NtQueryInformationProcess:
Funkcija vrne informacije o trenutnem procesu, iz teh informacij pa se da razbrati, ali se proces izvaja znotraj razhroščevalnika ali ne. Za zaznavanje razhroščevalnika nastavimo parameter ProcessInformationClass na vrednost ProcessDebugPort (vrednost 7), funkcija nam vrne vrednost različno od 0, če se proces izvaja v razhroščevalniku.

DebugBreak:

Funkcija povzroči prekinitveno izjemo v procesu. V primeru, da se proces ne izvaja v razhroščevalniku, obravnavanje izjeme prevzame SEH (angl. structure exception handler). V nasprotnem primeru obravnavanje izjeme prevzame razhroščevalnik. Prav to lahko izkoristimo za zaznavanje razhroščevalnika, saj če se ob izvedbi funkcije ne generira izjema, pomeni, da se proces izvaja v razhroščevalniku. Primer kode prikazuje izpis 4.4.

```
__try{
    DebugBreak();
    MessageBoxA(0,''Razhroščevalnik zaznan'','''',0);
}
__except(EXCEPTION_EXECUTE_HANDLER){
    MessageBoxA(0,''Razhroščevalnik ni zaznan'','''',0);
}
```

Izpis 4.4: Zaznavanje razhroščevalnika s pomočjo DebugBreak API klica

4.2.2 Zaznavanje prekinitvenih točk

Razhroščevalnik uporablja dve vrsti prekinitvenih točk, strojne (angl. hardware breakpoint) in programske (angl. software breakpoint) [1].

Strojne prekinitvene točke:

Ob uporabi strojnih prekinitvenih točk se nastavijo vrednosti v razhroščevalnih registrih (tabela 2.6). Do registrov nimamo neposrednega dostopa iz uporabniškega načina, lahko pa pridemo do njihovih vrednosti s funkcijo GetThreadContext, ki nam vrne strukturo CONTEXT. Struktura vsebuje informacije o stanju registrov v niti procesa ter še nekatere druge informacije, med drugim tudi vrednosti razhroščevalnih registrov. Prekinitvene točke lahko pobrišemo s tem, da vrednosti registrov v strukturi postavimo na 0 in kličemo funkcijo SetThreadContext, ki nastavi nove vrednosti.

Programske prekinitvene točke:

Za programske prekinitvene točke se uporabljajo posebni strojni ukazi, ki jih razhroščevalnik umesti namesto obstoječih ukazov. Večina razhroščevalnikov uporablja ukat INT 3 (operacijska koda je 0xCC), Ta

beseda se zamenja z obstoječo besedo na naslovu, kjer se postavi prekinitvena točka. Za zaznavanje prekinitvenih točk se najpogosteje uporablja rutine za preverjanje integritete programske kode. V ta namen se lahko uporabi preverjanje ciklične redundance [18], saj se ob postavitvi prekinitvene točke spremeni sama operacijska koda. Izpis 4.5 prikazuje izvorno kodo, ki preverja ciklično redudanco. Funkcija genCrc32Table napolni pomožno tabelo, ki jo funkcija checkCrc32 uporabi pri računanju redudance. Če je redudanca 0, potem prekinitvena točka ni postavljena.

```
unsigned int crcTable[256];

void genCrc32Table(){
    unsigned int polynom,crc;
    int i,j;

    polynom = 0xEDB88320;
    for(i = 0; i < 256; i++){
        crc = (unsigned int)i;
        for(j = 8; j > 0; j--){
            if(crc & 1)
                crc = (crc >> 1) ^ polynom;
            else
                crc = crc >> 1;
        }
    }
}

int checkCrc32(unsigned char *buffer, uint buffLength, uint validCrc32){
    uint crc32,i;
       crc32 = 0xFFFFFFFF;

    for(i = 0; i < buffLength; i++){
        crc32 = (((crc32 >> 8) & 0x00FFFFFF) ^
                crcTable[((unsigned char)crc ^ *buffer++) & 0xFF]);
    }

    return (crc32 ^ 0xFFFFFFFF) ^ validCrc32;
}
```

Izpis 4.5: Zaznavanje prekinitvenih točk s CRC32 algoritmom

4.2.3 Zaznavanje s časovno razliko

Pri zaznavanju razhroščevalnikov si lahko pomagamo tudi z merjenjem časovnih razlik izvajanja programske kode. Za ta namen lahko uporabimo procesorski visokoločljivostni števec, ali pa uporabimo API klic GetTickCount. Uporaba GetTickCount funkcije je manj zanesljiva, saj funkcija vrača vrednost milisekund, in je za manjše bloke kode neuporabna.

Visokoločljivostni števec:
Procesor vsebuje 64-bitni visokoločljivostni števec, ki začne šteti cikle ure ob prekinitvei *reset* v procesorju. Za dostop do števca lahko uporabimo ukaz RDTSC (izpis 4.6), ta nam vrne 64-bitno vrednost v registra EAX in EDX. Te vrednosti lahko uporabimo za zaznavanje števila urnih ciklov, ki jih je porabila programska koda, da se je izvedla. Za branje vrednosti števca lahko uporabimo tudi API klic QueryPerformanceCounter (izpis 4.7).

```
DWORD startValue;
DWORD endValue;

__asm{
    push eax
    push edx
    rdtsc
    mov startValue,eax
    pop edx
    pop eax
}

// KODA, KATERI MERIMO IZVAJANJE

__asm{
    push eax
    push edx
    rdtsc
    mov endValue,eax
    pop edx
    pop eax
}

if(40 < (endValue − startValue)){
```

```
        MessageBoxA(0,"Razhroščevalnik zaznan","",0);
    }
    else{
        MessageBoxA(0,"Razhroščevalnik ni zaznan","",0);
    }
```

Izpis 4.6: Uporaba visokoločljivostnega števca za zaznavanje razhroščevalnika

```
    int i = 0;
    LARGE_INTEGER start;
    LARGE_INTEGER end;

    QueryPerformanceCounter(&start);
    i = 500;
    i += i; // koda, ki jo želimo meriti
    i *= i;
    QueryPerformanceCounter(&end);

    if((end.LowPart - start.LowPart) > 50){
        MessageBoxA(0,"Razhroščevalnik zaznan","",0);
    }
    else{
        MessageBoxA(0,"Razhroščevalnik ni zaznan","",0);
    }
```

Izpis 4.7: Uporaba QueryPerformanceCounter API klica za zaznavo razhroščevalnika

4.3 Metode za onemogočanje analize obratnega zbirnika

Obratni zbirniki uporabljajo dva tipa algoritmov za analizo, linearne in rekurzivne. Za onemogočanje algoritmov in za samo obfuskacijo programske kode se uporabljajo metode, kot so uporaba odvečne kode (angl. junk code), samoreplikacijske koda (angl. self modifying code) ali obfuskacija z navideznim strojem (angl. virtual machine obfuscation). Vse te metode otežijo tudi proces razhroščevanja, saj je veliko težje slediti poteku odvečne ali samoreplikacijske

kode. Ob uporabi navideznega stroja se sama analiza tudi precej oteži, saj je potrebno analizirati navidezni stroj in kodo, ki jo navidezni stroj izvaja.

4.3.1 Odvečna koda

Z odvečno kodo skušamo onemogočiti analizo strojne kode, kar pripelje do napačnega prikaza zbirne kode v obratnem zbirniku. Linearni algoritem je lažje onemogočiti, saj uporablja za dekodiranje ukaza samo njegovo velikost, ne pa tudi poteka same kode. Za onemogočanje se uporabljajo skočni ukazi (JMP, JZ, JNZ itd.) ter ukazi za klic podprogramov (CALL). Vsi ti ukazi spremenijo kazalec na naslednji ukaz, ki se bo izvedel [1, 19]. Izpis 4.8 prikazuje odvečno kodo, ki onemogoči linearni algoritem.

```
__asm{
    _emit 0xEB //
    _emit 0x01 // 0xEB 0x01: JMP $ + 3
    _emit 0xB8 // op. koda za MOV EAX, imm32
}

MessageBoxA(0,"Uporaba odvečne kode","",0);
```

Izpis 4.8: Odvečna koda za onemogočanje linearnega algoritma

Prvi dve besedi sestavljata ukaz JMP, ta ukaz poveča kazalec na naslednji ukaz za 3. Naslednja beseda je operacijska koda za ukaz MOV, ki v EAX register shrani 32-bitno vrednost, vendar se ta ne bo nikoli izvedel, ker se pred njim zmeraj izvede brezpogojni skok. Ker linearni algoritem ne upošteva samega poteka zbirne kode, bo prepoznal JMP in MOV ukaza. Izpis 4.9 prikazuje napačno analizirano zbirno kodo.

```
00401000: JMP 00401003
00401002: MOV EAX, 8468006A
00401007: XOR EAX, [EAX + 00]
0040100A: PUSH 00403018 ; "Uporaba odvečne kode"
0040100F: PUSH 00
00401011: CALL MessageBoxA
```

Izpis 4.9: Napačno analizirana koda v Softice razhroščevalniku

Rekurzivni algoritem z odvečno kodo iz izpisa 4.8 nima težav, saj upošteva še potek same kode. IDA prepozna JMP ukaz, operacijsko kodo za ukaz MOV

pa označi kot dodatno besedo. Analizo nadaljuje na naslovu, na katerega kaže JMP ukaz. Analizo zbirne kode prikazuje izpis 4.10.

```
.text:00401000 jmp short loc_401003
.text:00401002 db 0B8h
.text:00401003 loc_401003:
.text:00401003 push 0 ; uType
.text:00401005 push offset Caption ; lpCaption
.text:0040100A push offset Text ; "Uporaba odvečne kode"
.text:0040100F push 0 ; hWnd
.text:00401011 call ds:__imp__MessageBoxA@16 ; MessageBoxA(x,x,x,x)
```

Izpis 4.10: Pravilno analizirana koda obratnega zbirnika IDA

Za onemogočanje rekurzivnega algoritma se uporablja odvečna koda s pogojnimi skoki. Rekurzivni algoritem upošteva naslov pogojnega skoka kot del programa, ki ga bo analiziral naslednjega. V primeru, da se pogojni skok nikoli ne izvrši, analizirana koda lahko ni prava, če je naslednji ukaz, ki sledi pogojnemu skoku, brezpogojni skok, ki spremeni sam potek strojne kode. Izpis 4.11 prikazuje primer odvečne kode, ki onemogoči rekurzivni algoritem. Najprej se shrani vrednost EAX registra, vrednost EAX se z ukazom XOR postavi na 0 ter z INC ukazom poveča za 1. Rezultat te operacije je zmeraj vrednost 1 v EAX registru, ZF v EFLAGS registru (tabela 2.7) pa je postavljen na 0. Sledi POP ukaz, ki shranjeno vrednost vrne v EAX register, kar pa ne spremeni vrednosti v EFLAGS registru. Naslednji dve besedi predstavljata pogojni skok JZ, ki se ne bo nikoli izvedel, saj ima ZF bit vrednost 0. IDA bo analizirala zbirno kodo na naslovu, kamor kaže skok JZ (v našem primeru na operacijsko kodo za MOV ukaz), čeprav se zbirna koda ne bo nikoli izvedla, kar privede tudi do napačne analize zbirne kode.

```
__asm{
    push eax
    xor eax,eax
    inc eax
    pop eax
    _emit 0x74 //
    _emit 0x02 // JZ $ + 4
    _emit 0xEB //
    _emit 0x01 // JMP $ + 3
    _emit 0xB8 // MOV EAX, imm32
```

```
}

MessageBoxA(0,"Uporaba odvečne kode","",0);
```

Izpis 4.11: Odvečna koda za onemogočanje rekurzivnega algoritma

Izpis 4.12 prikazuje napačno analizirano kodo. Obratni zbirnik upošteva vejo programa, ki se ne bo nikoli izvedla (naslov 0x00401009), kar privede do napačne analize zbirne kode. Ukaza "mov eax, 8468006Ah" in "xor eax, [eax+0]" namreč nista pravilna, pravilna ukaza bi bila "push 0" in "push offset Caption", ki jih prikazuje izpis 4.10.

```
.text:00401000 push eax
.text:00401001 xor eax, eax
.text:00401003 inc eax
.text:00401004 pop eax
.text:00401005 jz short loc_401009
.text:00401007 jmp short near ptr loc_401009+1
.text:00401009 loc_401009:
.text:00401009 mov eax, 8468006Ah
.text:0040100E xor eax, [eax+0]
.text:00401011 push offset Text ; "Uporaba odvečne kode"
.text:00401016 push 0 ; hWnd
.text:00401018 call ds:__imp__MessageBoxA@16 ; MessageBoxA(x,x,x,x)
```

Izpis 4.12: Napačno analizirana koda obratnega zbirnika IDA

4.3.2 Samoreplikacijska koda

Samoreplikacijska koda spreminja operacijsko kodo med samim izvajanjem, kar močno oteži izvajanje obratnega inženiringa, onemogoči pa tudi obratne zbirnike, saj se programska koda pravilno dekodira šele med samim izvajanjem. Kodo je težje tudi modificirati, saj obstaja možnost, da se ob morebitni modifikaciji koda ne bo dekodirala pravilno, kar bo privedlo do sesutja programa. Slabost uporabe samoreplikacijske kode je v tem, da sama koda vpliva na procesorski predpomnilnik. Procesor za izvajanje kode v predpomnilnik naloži programsko kodo, ki jo bo izvedel. Ker se programska koda med samim izvajanjem spreminja, se mora ob vsaki spremembi kode naložiti v predpomnilnik novo generirana koda, kar posledično pomeni daljši čas izvajanja. V kombinaciji s samoreplikacijsko kodo se uporabljajo tudi tehnike polimorfizma

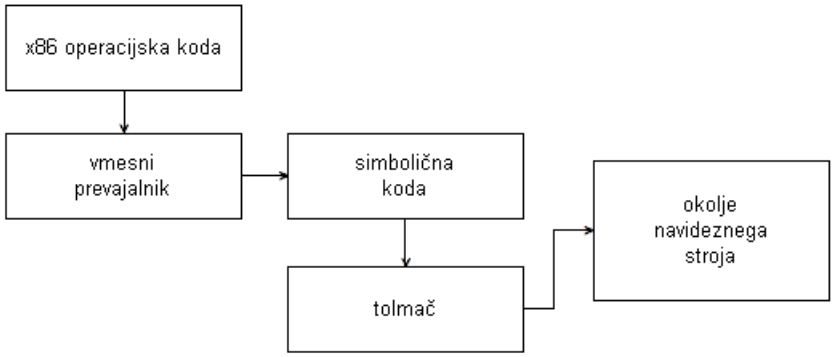
ter metamorfizma [1]. Polimorfizem je tehnika, kjer samoreplikacijska koda ob vsaki izvedbi drugače kodira in dekodira zbirno kodo. Polimorfična koda uporablja polimorfično rutino, ki omogoča drugačno spremembo kode vsakič, ko se le-ta izvede. Tako dobimo programske sklope, ki so si različni, vendar se vsi dekodirajo v enako zbirno kodo. Metamorfizem je tehnika, kjer se semantika samoreplikacijske kode spreminja z vsako izvedbo. Metamorfična koda uporablja posebno metodo, ki omogoča generiranje različne zbirne kode, izvedba oziroma rezultat pa je vedno isti. Tehniki se uporabljata tudi pri pisanju virusov ter črvov, saj onemogočajo zaznavanje zlonamerne kode s strani protivirusnih programov. Po navedbah izdelovalca protivirusnih programov Kaspersky Lab se tehnika polimorfizma uporablja v enem izmed najbolj razširjenih virusov, imenovanem "Virus.Win32.Virut.ce"[1].

O odvečni in samoreplikacijski kodi bomo govorili še v poglavjih 5 in 6, saj naša rešitev, ki je opisana v poglavju 7, temelji na teh dveh metodah.

4.3.3 Uporaba navideznaga stroja

Uporaba navideznega stroja (slika 4.1) je zelo učinkovita metoda tako za onemogočanje obratnega zbirnika kot tudi razhroščevalnika [1]. Najprej vmesni prevajalnik pretvori strojno kodo v simbolično, strojno kodo pa premakne v navidezni stroj. Ob zagonu programa se začne izvajati navidezni stroj, ki s pomočjo vgrajenega tolmača dekodira simbolično kodo in na podlagi le-te izvede zaporedje strojnih ukazov. Zaporedje strojnih ukazov je v navideznem stroju pomešano, zato je potrebno najprej analizirati zgradbo navideznega stroja. Simbolične kode ni mogoče analizirati z obratnim zbirnikom ali razhroščevalnikom, saj je v celoti odvisna od zasnove navideznega stroja. Ker celotna zaščita temelji na nepoznavanju delovanja navideznega stroja, v sam navidezni stroj dodatno vgradimo še odvečno in samoreplikacijsko kodo in tako dodatno otežimo samo analizo le-tega.

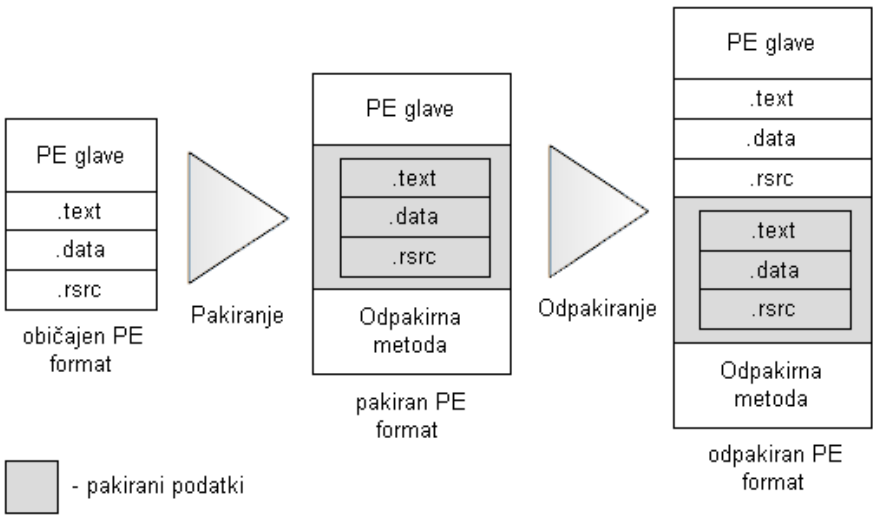
[1] `http://www.securelist.com/en/analysis/204792122/Review_of_the_Virus_Win32_Virut_ce_Malware_Sample`



Slika 4.1: Diagram izvajanja kode v navideznem stroju

4.4 Primerjava

Metod nasprotno-obratnega inženiringa se poslužuje veliko programov, katerih cilj je zaščita programske opreme. Večina programov je PE pakirnikov, ki zapakirajo PE format EXE ali DLL datoteke. Običajno tudi izbrišejo večino API klicev iz uvozne tabele, imena uvoznih funkcij pa zapakirajo med ostale pakirane podatke. Slika 4.2 prikazuje tipično delovanje pakirnikov.



Slika 4.2: Tipično delovanje PE pakirnika

Pakirnik zapakira sekcije in doda odpakirno rutino PE formatu. Rutina odpakira sekcije, iz odpakiranih imen uvoznih funkcij pa zgradi uvozno tabelo, ki jo doda odpakiranem PE formatu. Na koncu odpakirna rutina preusmeri izvajanje na izvorno vstopno točko programa (angl. original entry point ali OEP). Na tej točki je v pomnilniku naložen odpakiran PE format programa, ki ga izvozimo na disk. Izvoženemu PE formatu nato dodamo uvozno tabelo, tako dobimo nezaščiten delujoč program, nad katerim izvajamo obratni inženiring. Odpakirna rutina navadno vsebuje metode za zaznavanje razhroščevalnikov in odvečno kodo za onemogočanje obratnega zbirnika. Tako se oteži sama analiza odpakirne rutine ter s tem možnost, da se iz pomnilnika izvozi odpakiran program. Skoraj za vsak pakirnik se dobijo namenski programi, ki odpakirajo določeno verzijo pakirnika. To je precej nepraktično, saj so potrebni različni programi za različne verzije istega pakirnika. Na spletu se dobijo tudi splošni odpakirniki (angl. generic unpackers), ki odpakirajo več različnih tipov pakirnikov. V diplomski nalogi smo za testiranje pakirnikov uporabili naslednje splošne odpakirnike:

- "Universal PE Unpacker" je vtičnik za vgrajeni razhroščevalnik v IDA obratnem zbirniku. Vtičnik deluje tako, da poskuša ob zagonu pakiranega programa postaviti prekinitveno točko na API klic GetProcAddress v uvozni tabeli. Klic vrne naslov izvozne funkcije za podano dinamično knjižnico, pakirniki pa ga uporabljajo pri rekonstrukciji uvozne tabele, ki jo generirajo med odpakiranjem programa [20]. Izvorno vstopno točko določimo tako, da v vtičniku določimo začetni in končni naslov, kjer bi se le-ta lahko nahajala. Ker neplačljiva različica obratnega zbirnika tega vtičnika ne vsebuje, smo za testiranje uporabili zadnjo verzijo predstavitvene različice IDA Pro 6.2[2].

- "OllyDump"[3] je vtičnik za razhroščevalnik OllyDbg, ki smo ga opisali v poglavju 3. Vtičnik išče izvorno vstopno točko tako, da izvaja program korak za korakom. Ko se izvajanje programske kode preusmeri v drugo sekcijo, vtičnik to zazna kot izvorno vstopno točko. Večina pakirnikov ima namreč ločeno odpakirno rutino in pakirane podatke v različnih sekcijah. Ko odpakirna rutina preusmeri izvajanje na odpakirano kodo, se le-ta nahaja v drugi sekciji, kar vtičnik zazna. Poleg tega vtičnik vsebuje tudi funkcijo za izvoz PE formata iz pomnilnika na disk, kar olajša sam proces odpakiranja [21].

[2] `http://www.hex-rays.com/products/ida/support/download_demo.shtml`
[3] `http://www.openrce.org/downloads/details/108/OllyDump`

- "Polyunpack"[4] je prav tako vtičnik za OllyDbg, za analizo pa uporablja kombinacijo statične in dinamične analize. Vtičnik namreč primerja kodo, ki jo je izdelal obratni zbirnik v razhroščevalniku, s kodo, ki se trenutno izvaja. Če se kodi na določenih naslovih ne ujemata, pomeni, da je prišlo do odpakiranja programske kode. Vtičnik postavi prekinitvene točke na te naslove, aktivirajo pa se, preden se odpakirani del kode začne izvajati [22].

- "Quick Unpacker 2.2"[5] je samostojni program, ki ima vgrajen obroč 0 razhroščevalnik (glej poglavje 2.2) za odpakiranje programov. Poleg tega uporablja tudi transparentno prestrezanje API klicev (angl. API hooking), kar se s pridom izkorišča za onemogočanje metod nasprotno-obratnega inženiringa. S transparentnim prestrezanjem prestrežemo API klic IsDebuggerPresent ter mu spremenimo vrednost, ki jo vrača. Tako lahko zaobidemo zaznavanje razhroščevalnika. Izvorno vstopno točko moramo sami določiti, lahko pa uporabimo nekaj vgrajenih vtičnikov za iskanje le-te [23].

Ta orodja smo testirali nad pakirniki Upx 3.07[6], ASProtect 1.64[7], PeLock 1.06[8] in Enigma Protector 3.10[9]. Z vsemi pakirniki smo zapakirali preprost program, ki smo ga napisali v C-ju in ga prikazuje izpis 4.13.

```
int main(int argc, char *argv[]){
    MessageBoxA(0,(LPCSTR)"Testni primer",(LPCSTR)"Test",0);
    return 0;
}
```

Izpis 4.13: Izvorna koda programa za testiranje pakirnikov

Izpis 4.14 prikazuje zbirno kodo main() funkcije, tabela 4.1 pa prikazuje spremembe, ki so jih pakirniki naredili v PE formatu testnega programa. Vsebuje podatke o številu sekcij, velikost datoteke po pakiranju in še nekaj drugih informacij. Vse podatke smo pridobili z programom LordPE, s katerim smo analizirali PE format testnega programa. Omeniti velja, da je polje "ImageBase" v neobvezni glavi testnega programa nastavljeno na naslov 0x00400000.

[4] http://polyunpack.cc.gt.atl.ga.us/polyunpack.zip
[5] http://www.woodmann.com/collaborative/tools/index.php/Quick_Unpack
[6] http://upx.sourceforge.net/
[7] http://www.aspack.com/asprotect.html
[8] http://www.techspot.com/downloads/2886-pelock.html
[9] http://www.enigmaprotector.com/en/downloads.html

```
.text:00401000 push 0 ; uType
.text:00401002 push offset Caption ; "Test"
.text:00401007 push offset Text ; "Testni primer"
.text:0040100C push 0 ; hWnd
.text:0040100E call ds:MessageBoxA
.text:00401014 xor eax, eax
.text:00401016 retn
```

Izpis 4.14: Zbirna koda main() funkcije testnega programa

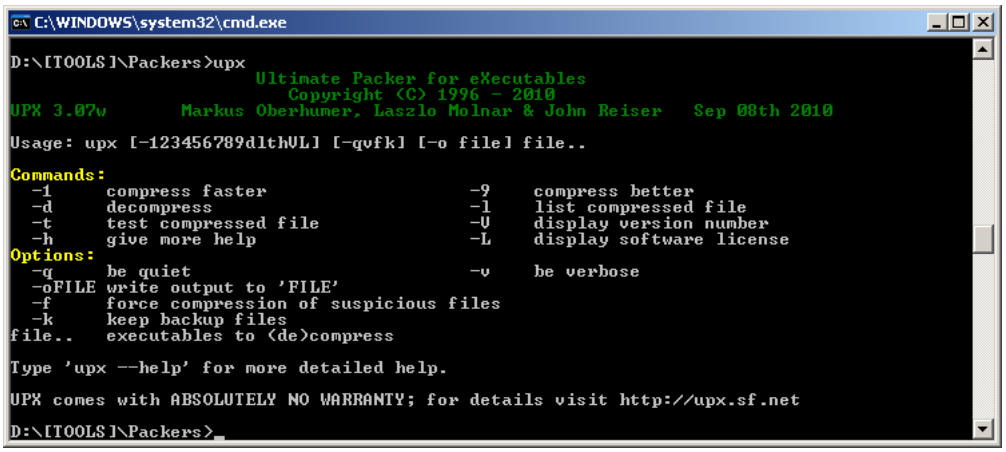
Pakirnik	Velikost (v besedah)	Število sekcij	Sekcije	Opombe
nepakiran program	7680	5	.text, .rdata, .data, .rsrc, .reloc	Sekcija .text je označena, da vsebuje izvajalno kodo, ostale ne. Naslov vstopne točke je 0x004012C8.
Upx 3.07	5120	3	UPX0, UPX1, .rsrc	Sekciji UPX0 in UPX1 vsebujeta izvajalno kodo, naslov vstopne točke je nastavljen na 0x00407940.
ASProtect 1.64	144896	7	.data, .adata, .rsrc, vse ostale sekcije nimajo imena	Vse sekcije so označene, da vsebujejo izvajalno kodo. Naslov vstopne točke: 0x00401000.
PeLock 1.06	21504	5	.rsrc in pet sekcij z imenom .pelock	Nobena sekcija ni označena, da vsebuje izvajalno kodo, vstopna točka je nastavljena na naslov 0x00405150.

Enigma Protector 3.10	759296	8	.rsrc, .data in 6 sekcij brez imena	Vse sekcije so označene, da vsebujejo izvajalno kodo. Vstopna točka se nahaja na naslovu 0x004012E9.

Tabela 4.1: Spremembe PE formata testnega programa, pakiranega z različnimi pakirniki

4.4.1 Upx 3.07

Upx 3.07 je odprtokodni pakirnik, ki se ga poganja v konzoli. Glavna prednost pakirnika pred ostalimi je velika kompresija. Na voljo imamo 9 možnosti kompresije, ki jih lahko izberemo iz konzolnega vmesnika (slika 4.3), preko katerega pa ne moremo nastaviti uporabe metod nasprotno-obratnega inženiringa, saj jih pakirnik ne uporablja.



Slika 4.3: Konzolni vmesnik Upx 3.07 pakirnika

Pri uporabi Upx pakirnika smo uporabili opcijo -9 (največja kompresija), nato smo pakiran program naložili v IDA obratni zbirnik. Ugotoviti moramo namreč približno lokacijo izvorne vstopne točke, saj večina prej opisanih avtomatskih odpakirnikov ta podatek potrebuje. Izpis 4.15 prikazuje zbirno

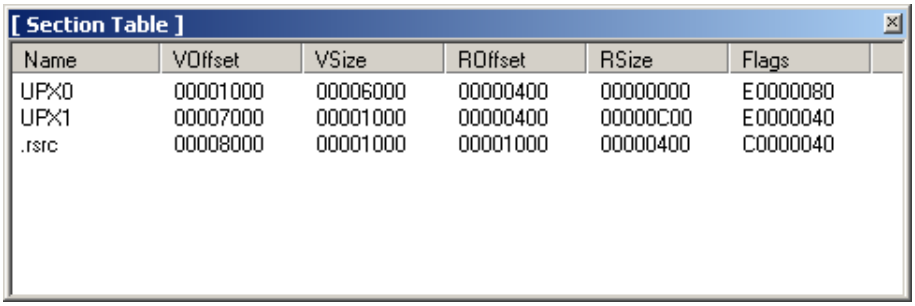
kodo, ki se nahaja na vstopni točki. Na začetku se nahaja PUSHAD ukaz, ki shrani vrednosti vseh registrov, nato se v register ESI naloži kazalec na naslov 0x00407000, v EDI pa se naloži naslov 0x00401000. Nato se začne proces odpakiranja.

```
UPX1:00407940 PUSHAD
UPX1:00407941 MOV ESI, OFFSET DWORD_407000
UPX1:00407946 LEA EDI, [ESI−6000H]
UPX1:0040794C PUSH EDI
UPX1:0040794D JMP SHORT LOC_40795A
UPX1:00407950 LOC_407950:
UPX1:00407950 MOV AL, [ESI]
UPX1:00407952 INC ESI
UPX1:00407953 MOV [EDI], AL
UPX1:00407955 INC EDI
UPX1:00407956 LOC_407956:
UPX1:00407956 ADD EBX, EBX
UPX1:00407958 JNZ SHORT LOC_407961
UPX1:0040795A LOC_40795A:
UPX1:0040795A MOV EBX, [ESI]
UPX1:0040795C SUB ESI, 0FFFFFFFCH
UPX1:0040795F ADC EBX, EBX
UPX1:00407961 LOC_407961:
UPX1:00407961 JB SHORT LOC_407950
UPX1:00407963 MOV EAX, 1
```

Izpis 4.15: Del zbirne kode vstopne točke Upx 3.07 odpakirne rutine

Če primerjamo kazalce v EDI in ESI registrih s sliko sekcij na sliki 4.4, vidimo, da kazalec v ESI registru kaže na začetek sekcije UPX1, kazalec v EDI registru pa na začetek sekcije UPX0. To ugotovimo tako, da od kazalcev v registrih odštejemo vrednost "ImageBase" polja v neobvezni glavi, ki ima vrednost 0x00400000. Tako dobimo vrednosti 0x00007000 in 0x00001000, ki sta enaki navideznim odmikom UPX1 in UPX0 sekcij v PE formatu.

Iz krajše analize odpakirne metode (zbirne kode nismo umestili v diplomsko delo zaradi njene dolžine) in iz prej ugotovljenih podatkov, lahko sklepamo, da odpakirna metoda najprej odpakira program v sekcijo UPX0, potem pa v to sekcijo preusmeri izvajanje programa. Tako lahko rečemo, da je izvorna vstopna točka programa nekje med naslovi 0x00401000 in 0x00407000. Tabela 4.2 prikazuje rezultate avtomatskih odpakirnikov nad testnim programom.



Slika 4.4: Sekcije v testnem programu, pakiranim z Upx pakirnikom

Odpakirnik	Uspešno odpakiran program	Se program izvaja	Okvirni čas odpakiranja (v sekundah)	Čas, potreben za zaznavo razhroščevalnika (v sekundah)
Universal PE Unpacker	da	da	3	/
OllyDump	da	da	6	/
Polyunpack	da	da	1170	/
Quick Unpacker 2.2	da	da	3	/

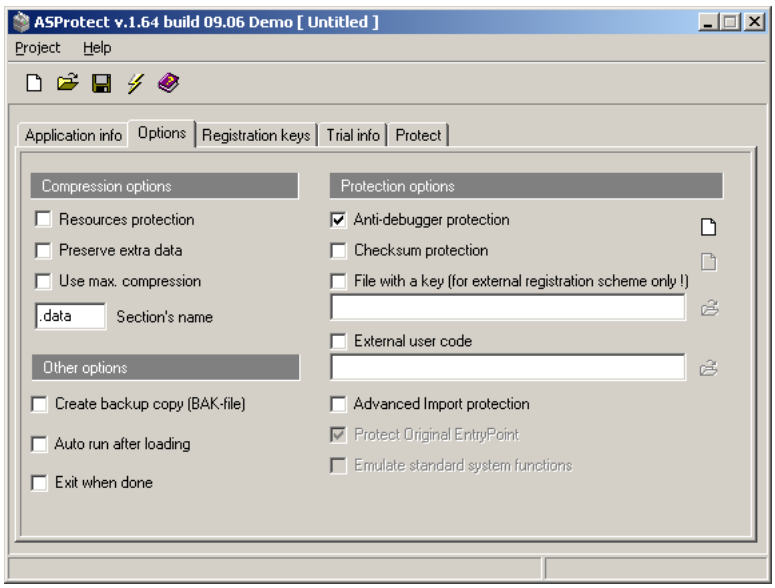
Tabela 4.2: Rezultati avtomatskih odpakirnikov za Upx 3.07 pakirnik

Iz tabele 4.2 je razvidno, da večina odpakirnikov precej hitro odpakira program, le Polyunpack porabi v primerjavi z drugimi precej več časa. To gre pripisati načinu delovanja (program izvaja korak za korakom). Čas smo uspeli prepoloviti, če vtičnik v večjih zankah ustavimo in pustimo, da se zanka normalno konča samo znotraj razhroščevalnika. Vsem odpakirnikom je uspelo odpakirati program, saj odpakirna rutina ne uporablja nobenih metod za zaznavanje razhroščevalnikov.

4.4.2 ASProtect 1.64

ASProtect 1.64 je komercialni pakirnik, dobi pa se ga tudi v preizkusni različici. Preizkusna različica vsebuje poleg standardnega pakiranja še veliko drugih možnosti, kot recimo uporaba kriptografije z javnim ključem, časovno omejitev delovanja programa in še nekatere druge.

Pri našem testiranju smo uporabili klasično pakiranje z uporabo metod za zaznavanje razhroščevalnikov ter zaščito izvorne vstopne točke. Slika 4.5 prikazuje grafični vmesnik pakirnika z izbranimi možnostmi pakiranja. Pakiran program smo naložili v obratni zbirnik in analizirali kodo na vstopni točki, ki jo prikazuje izpis 4.16. Iz izpisa in slike 4.6 vidimo, da se vstopna točka programa začne v prvi sekciji, koda, ki se izvede, pa preusmeri izvajanje na naslov 0x00406001, ki se nahaja v .data sekciji. To je tipičen trik za onemogočanje avtomatskih odpakirnikov, kot je OllyDump.



Slika 4.5: Grafični vmesnik ASProtect 1.64 pakirnika

```
SEG000:00401000 PUSH OFFSET SUB_406001
SEG000:00401005 CALL NULLSUB_1
SEG000:0040100A RETN
SEG000:0040100B NULLSUB_1 PROC NEAR
SEG000:0040100B RETN
SEG000:0040100B NULLSUB_1 ENDP
```

Izpis 4.16: Zbirna koda vstopne točke ASProtect odpakirne rutine

Po krajši analizi odpakirne rutine smo ugotovili, da vsebuje poleg odvečne kode tudi samoreplikacijsko kodo. Izpis 4.17 predstavlja zbirno kodo pred izvedbo ukaza REP MOVSB, izpis 4.18 pa po izvedbi. Ukaz prenese besede iz

pomnilnika, na katerega kaže register ESI, v pomnilnik, na katerega kaže EDI register. Število besed za prenos vsebuje ECX register. Zbirna koda se v izpisu 4.18 začne spreminjati na naslovu 0x0045B101.

```
.DATA:0045B0EB REP MOVSB
.DATA:0045B0ED MOV EAX, SS:DWORD_442975[EBP]
.DATA:0045B0F3 PUSH 8000H
.DATA:0045B0F8 PUSH 0
.DATA:0045B0FA PUSH EAX
.DATA:0045B0FB CALL SS:DWORD_44297D[EBP]
.DATA:0045B101 LEA ECX, [ESI]
.DATA:0045B103 TEST [ECX+2CH], EDX
.DATA:0045B106 INC ESP
.DATA:0045B107 POP ES
.DATA:0045B108 PUSH EAX
```

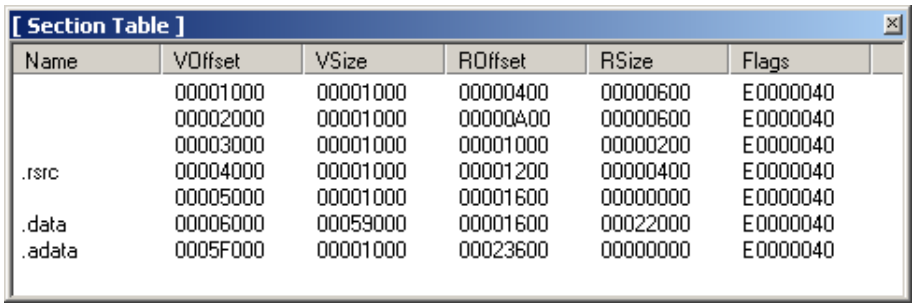
Izpis 4.17: Samoreplikacijska koda v ASProtect 1.64 odpakirni rutini - pred izvedbo ukaza REP MOVSB

```
.DATA:0045B0EB REP MOVSB
.DATA:0045B0ED MOV EAX, SS:DWORD_442975[EBP]
.DATA:0045B0F3 PUSH 8000H
.DATA:0045B0F8 PUSH 0
.DATA:0045B0FA PUSH EAX
.DATA:0045B0FB CALL SS:DWORD_44297D[EBP]
.DATA:0045B101 LEA EAX, [EBP+442C51H]
.DATA:0045B103 PUSH ECX
.DATA:0045B106 ADD [EAX−3DH], DL
.DATA:0045B107 PUSH EAX
.DATA:0045B108 RETN
```

Izpis 4.18: Samoreplikacijska koda v ASProtect 1.64 odpakirni rutini - po izvedbi ukaza REP MOVSB

Tabela 4.3 prikazuje rezultate avtomatskih orodij nad pakirniki. Pri testiranju prvih treh orodij je odpakirna metoda zaznala razhroščevalnik in po opozorilu prekinila izvajanje programa. Quick Unpacker se je izognil zaznavi razhroščevalnika, vendar je imel probleme pri zaznavanju izvorne prekinitvene točke. Za vsako sekcijo, ki smo jo izbrali, je ponudil namreč več kot 50 različnih izvornih vstopnih točk. Tudi ko smo uporabili izvorno vstopno točko 0x004012C8 iz tabele 4.1, je odpakirnik imel težave, saj se odpakirana koda ni ujemala z originalno, kar je povzročalo tudi nedelovanje (sesuvanje) programa.

To lahko vidimo, če primerjamo med seboj izpisa 4.14 in 4.19.



[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
	00001000	00001000	00000400	00000600	E0000040
	00002000	00001000	00000A00	00000600	E0000040
	00003000	00001000	00001000	00000200	E0000040
.rsrc	00004000	00001000	00001200	00000400	E0000040
	00005000	00001000	00001600	00000000	E0000040
.data	00006000	00059000	00001600	00022000	E0000040
.adata	0005F000	00001000	00023600	00000000	E0000040

Slika 4.6: Sekcije v testnem programu, pakiranim z ASProtect 1.64 pakirnikom

```
.TEXT:00401000 PUSH 0
.TEXT:00401002 PUSH OFFSET ATEST ; "Test"
.TEXT:00401007 PUSH OFFSET ATESTNIPRIMER ; "Testni primer"
.TEXT:0040100C PUSH 0
.TEXT:0040100E CALL NEAR PTR 0B50004H
.TEXT:00401013 MOV EAX, DS:3BC3C033H
.TEXT:00401018 OR EAX, OFFSET DWORD_403000
.TEXT:0040101D JNZ SHORT LOC_401021
.TEXT:0040101F REP RETN
.TEXT:00401021 LOC_401021:
.TEXT:00401021 JMP __REPORT_GSFAILURE
```

Izpis 4.19: Zbirna koda main() funkcije v odpakiranem programu z odpakirnikom Quick Unpacker 2.2

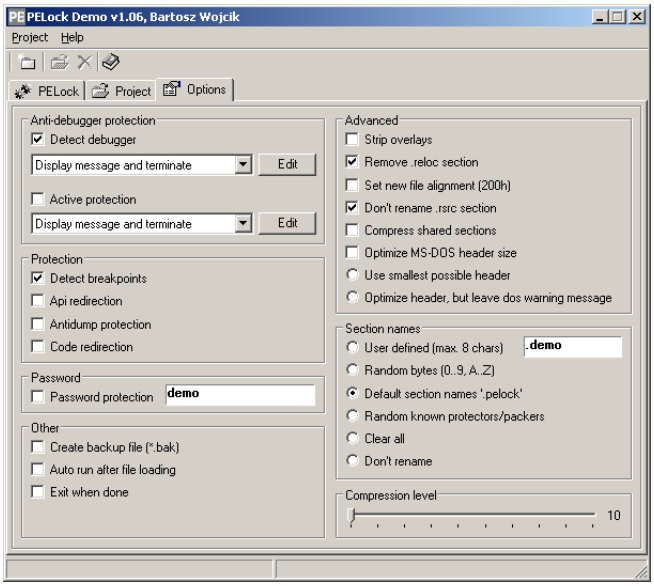
Odpakirnik	Uspešno odpakiran program	Se program izvaja	Okvirni čas odpakiranja (v sekundah)	Čas, potreben za zaznavo razhroščevalnika (v sekundah)
Universal PE Unpacker	ne	ne	/	6
OllyDump	ne	ne	/	2
Polyunpack	ne	ne	/	320

Quick Unpacker 2.2	delno	ne	3	/

Tabela 4.3: Rezultati avtomatskih odpakirnikov nad ASProtect 1.64 pakirnikom

4.4.3 PeLock 1.06

PeLock 1.06 (slika 4.7) je prav tako komercialni pakirnik, zato smo za našo primerjavo vzeli preizkusno različico. Ta poleg pakiranja omogoča še zaznavanje prekinitvenih točk in preusmeritev API klicev, ostale nastavitve pa niso na voljo v preizkusni različici. Poleg tega pakirnik vsebuje tudi zaglavno datoteko (angl. header file) z odvečno kodo. To kodo uporabimo za označevanje programskih sklopov, ki jih nato pakirnik dodatno zaščiti oz. kodira. Slednja možnost v preizkusni različici ni na voljo.



Slika 4.7: Grafični vmesnik PeLock 1.06 pakirnika

Slika 4.8 prikazuje sekcije pakiranega programa. Iz slike vidimo, da se vstopna točka programa začne v zadnji sekciji (glej tabelo 4.1). Izpis 4.20 prikazuje vstopno točko, ki je obfuskirana z odvečno kodo. IDA je napačno analizirala odvečno kodo, saj je koda od naslova 0x00405160 naprej napačna. Izpis 4.21 prikazuje zbirno kodo, ki se dejansko izvede znotraj razhroščevalnika. Obfuskacija je narejena s kombinacijo ukazov STC in JB. Prvi nastavi CF bit

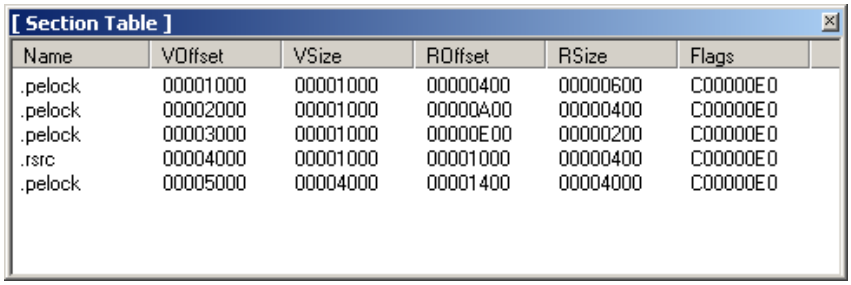
v EFLAGS registru (tabela 2.7) na 1, kar pomeni, da se bo nepogojni skok zmeraj izvedel. S tem onemogoči obratni zbirnik pri njegovi analizi.

```
.PELOCK:00405150 CMP ECX, EAX
.PELOCK:00405152 SHL ECX, 15H
.PELOCK:00405155 JMP SHORT LOC_405159
.PELOCK:00405157 DB 0CDH
.PELOCK:00405158 DB 20H
.PELOCK:00405159 LOC_405159:
.PELOCK:00405159 JMP SHORT LOC_40515D
.PELOCK:0040515B DB 0CDH
.PELOCK:0040515C DB 20H
.PELOCK:0040515D LOC_40515D:
.PELOCK:0040515D STC
.PELOCK:0040515E JB SHORT NEAR PTR LOC_405160+1
.PELOCK:00405160 LOC_405160:
.PELOCK:00405160 SBB CH, BL
.PELOCK:00405162 ADD AH, [EBX+3401EB91H]
.PELOCK:00405168 REPNE XOR CL, 92H
.PELOCK:0040516C SHR EAX, 0CH
.PELOCK:0040516F JMP SHORT LOC_405173
```

Izpis 4.20: Napačno analizirana zbirna koda na vstopni točki odpakirne rutine PeLock 1.06 z IDA obratnim zbirnikom

```
.PELOCK:0040515D LOC_40515D:
.PELOCK:0040515D STC
.PELOCK:0040515E JB SHORT LOC_405161
.PELOCK:00405160 DB 1AH
.PELOCK:00405161 LOC_405161:
.PELOCK:00405161 JMP SHORT LOC_405165
.PELOCK:00405163 DB 0A3H
.PELOCK:00405164 DB 91H
.PELOCK:00405165 LOC_405165:
.PELOCK:00405165 JMP SHORT LOC_405168
.PELOCK:00405167 DB 34H
.PELOCK:00405168 LOC_405168:
.PELOCK:00405168 REPNE XOR CL, 92H
.PELOCK:0040516C SHR EAX, 0CH
.PELOCK:0040516F JMP SHORT LOC_405173
```

Izpis 4.21: Pravilna zbirna koda na vstopni točki odpakirne rutine PeLock 1.06



Slika 4.8: Sekcije testnega programa, pakiranega s pakirnikom PeLock 1.06

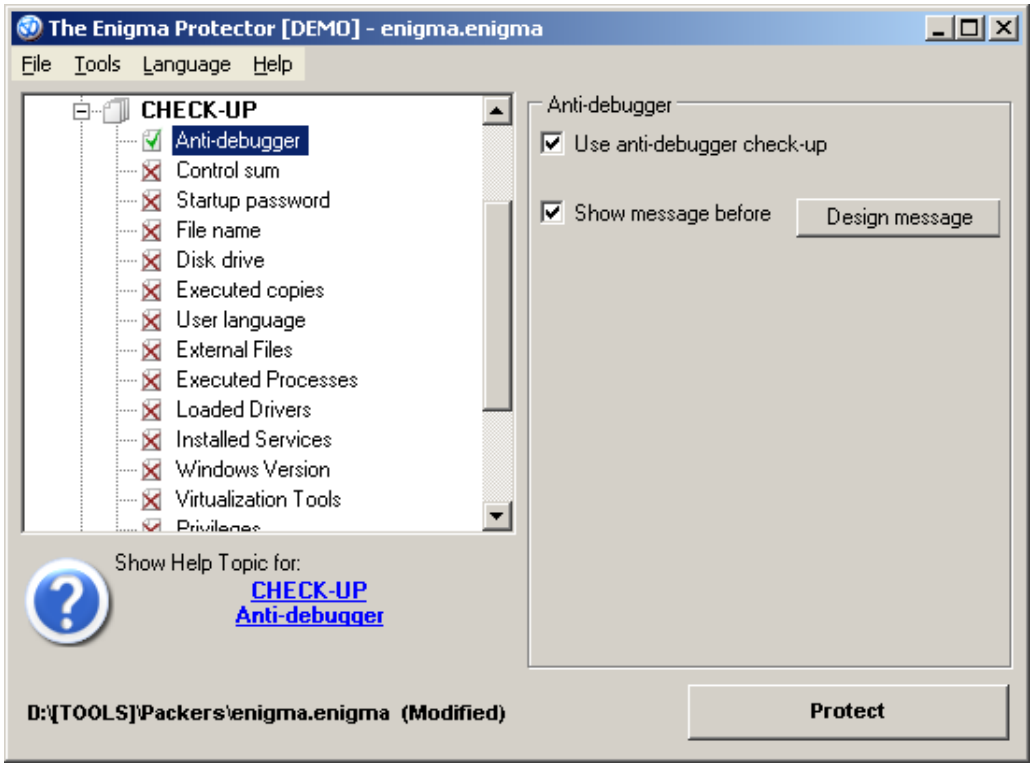
V tabeli 4.4 so prikazani rezultati avtomatskih odpakirnikov. Podobno kot pri prejšnem pakirniku je tudi ta zaznal oba razhroščevalnika, kjer smo poganjali vtičnike in prekinil proces odpakiranja. Quick Unpacker je v tem primeru uspešno odpakiral program, prav tako ni imel težav z zaznavanjem izvorne vstopne točke.

Odpakirnik	Uspešno odpakiran program	Se program izvaja	Okvirni čas odpakiranja (v sekundah)	Čas, potreben za zaznavo razhroščevalnika (v sekundah)
Universal PE Unpacker	ne	ne	/	5
OllyDump	ne	ne	/	4
Polyunpack	ne	ne	/	487
Quick Unpacker 2.2	da	da	3	/

Tabela 4.4: Rezultati avtomatskih odpakirnikov za PeLock 1.06 pakirnik

4.4.4 Enigma Protector 3.10

Enigma Protector 3.10 je komercialni pakirnik, ki uporablja poleg ostalih tehnik tudi tehniko navideznega stroja. Uporabili smo preizkusno različico, v kateri so poleg možnosti za zaznavanje razhroščevalnikov še preverjanje integritete zbirne kode, preverjanje naloženih gonilnikov (primerno za zaznavanje sistemskih monitorjev), virtualizacijo vstopne točke in še nekaj ostalih možnosti. Slika 4.9 prikazuje grafični vmesnik pakirnika Enigma Protector 3.10.



Slika 4.9: Grafični vmesnik pakirnika Enigma Protector 3.10

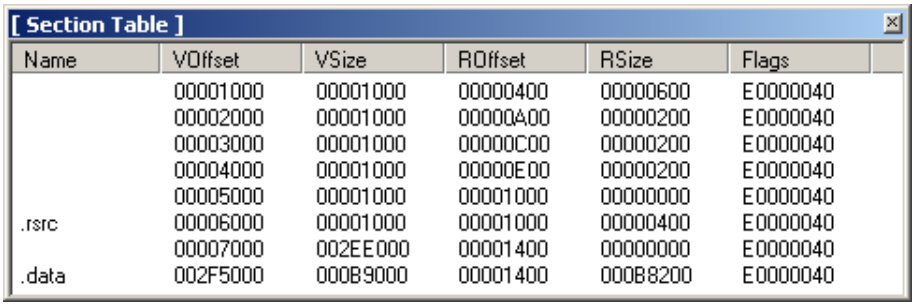
Pri tem pakirniku smo poleg opcije zaznavanja razhroščevalnikov uporabili še virtualizacijo programske vstopne točke. Izpis 4.22 prikazuje del odpakirne rutine, ki se izvaja v navideznem stroju. Vrednosti, ki se shranjujejo na sklad pred brezpogojnim skokom, določajo, kako se bo izvajala odpakirna rutina v navideznem stroju. Omeniti še velja, da se koda v izpisu 4.22 generira s pomočjo samoreplikacijske kode, tako da je koda dostopna le v pomnilniku. Za izvoz iz pomnilnika na disk smo uporabili orodje LordPE Deluxe (glej poglavje 3).

```
00588000 PUSH 0
00588005 JMP test_.00580F94
0058800A PUSH 5
0058800F JMP test_.00580F94
00588014 PUSH 0E
00588019 JMP test_.00580F94
0058801E PUSH 12
00588023 JMP test_.00580F94
```

```
00588028 PUSH 13
0058802D JMP test_.00580F94
00588032 PUSH 16
00588037 JMP test_.00580F94
```

Izpis 4.22: Primer kode navideznega stroja Enigma Protector 3.10 odpakirne rutine

Vsi štirje odpakirniki niso uspeli odpakirati testnega programa. Pri prvih treh je odpakirna rutina zaznala razhroščevalnik in prekinila proces odpakiranja. Quick Unpacker je imel kot pri ASProtect 1.64 težave pri zaznavanju izvorne vstopne točke. V našem primeru je ponudil več kot 65 izvornih vstopnih točk za prvo sekcijo (slika 4.10), v kateri naj bi se nahajala izvorna točka programa. Tudi če smo za izvorno vstopno točko uporabili naslov 0x004012C8 (glej tabelo 4.1), je bila odpakirana koda napačna.


[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
	00001000	00001000	00000400	00000600	E0000040
	00002000	00001000	00000A00	00000200	E0000040
	00003000	00001000	00000C00	00000200	E0000040
	00004000	00001000	00000E00	00000200	E0000040
	00005000	00001000	00001000	00000000	E0000040
.rsrc	00006000	00001000	00001000	00000400	E0000040
	00007000	002EE000	00001400	00000000	E0000040
.data	002F5000	000B9000	00001400	000B8200	E0000040

Slika 4.10: Sekcije testnega programa, pakiranega z Enigma Protector 3.10

Po krajši analizi odpakirane zbirne kode smo ugotovili, da je sicer main() funkcija uspešno odpakirana, vendar koda na vstopni točki generira izjemo, tako da do klica main() funkcije sploh ne pride. Rezultati odpakirnikov so prikazani v tabeli 4.5.

Odpakirnik	Uspešno odpakiran program	Se program izvaja	Okvirni čas odpakiranja (v sekundah)	Čas, potreben za zaznavo razhroščevalnika (v sekundah)
Universal PE Unpacker	ne	ne	/	5

OllyDump	ne	ne	/	4
Polyunpack	ne	ne	/	1487
Quick Unpacker 2.2	delno	ne	9	/

Tabela 4.5: Rezultati avtomatskih odpakirnikov za Enigma Protector 3.10 pakirnik

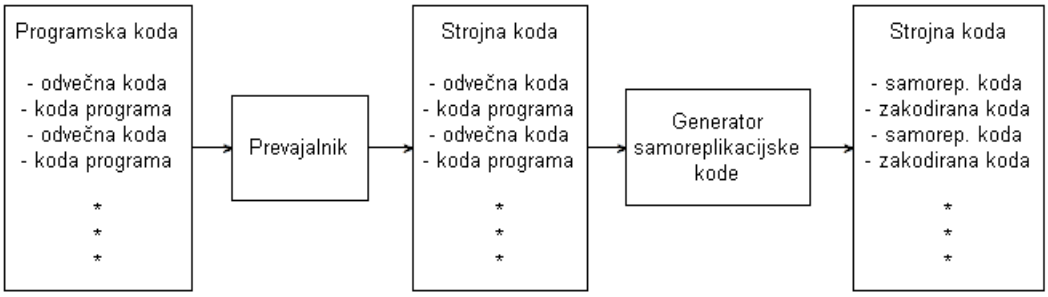
Iz rezultatov testiranja vidimo, da smo največ uspeha imeli z odpakirnikom Quick Unpack. Pri ostalih odpakirnikih je bil problem ta, da so metode nasprotno-obratnega inženiringa zaznale razhroščevalnik ter prekinile odpakiranje programa. Quick Unpacker se je sicer izognil zaznavam razhroščevalnika, vendar ni pravilno odpakiral rezultatov ASProtect in Enigma Protector pakirnikov. Rezultate bi se dalo izboljšati, če bi posamezne odpakirne rutine podrobneje osebno analizirali ter poskušali zaobiti metode za zaznavo razhroščevalnikov. Iz izpisov 4.17, 4.18 in 4.20 vidimo, da odpakirne rutine uporabljajo odvečno in samoreplikacijsko kodo, kar oteži in podaljša čas, potreben za analizo odpakirnih rutin.

Poglavje 5

Odvečna koda

Odvečna koda pri svoji izvedbi ne spremeni vrednosti registrov ali poteka same programske kode. Poleg uporabe za onemogočanje obratnega zbirnika ter oteženega razhroščevanja se lahko uporabi tudi kot digitalni tisk za preverjanje verodostojnosti aplikacije.

V diplomski nalogi smo se osredotočili predvsem na odvečno kodo, ki bo služila kot potreben prostor za samoreplikacijsko kodo in za onemogočanje obratnega zbirnika. Ideja je namreč, da se začetek in konec programskega bloka, ki se bo kodiral s pomočjo samoreplikacijske kode, označi z odvečno kodo, kot je prikazano na sliki 5.1. Odvečno kodo nato s pomočjo generatorja samoreplikacijske kode zamenjamo s samoreplikacijsko, programske bloke pa zakodiramo. Tako dobimo zakodiran program, katerega koda se dekodira tik preden se izvede in zakodira po njeni izvedbi.



Slika 5.1: Uporaba odvečne kode kot prostor za samoreplikacijsko kodo

Da zagotovimo nespremenjenost programa, je potrebno shraniti vse registre, ki jih bo uporabljala odvečna koda. Shraniti je potrebno tudi stanje

zastavic v EFLAGS registru, ali pa uporabljati zbirno kodo, ki ne vpliva na samo stanje. Za to je najlažje uporabiti programski sklad. Tabela 5.1 prikazuje ukaze, ki operirajo z skladom ter registri.

Zbirna koda	Opis
PUSH Reg32	Shrani vrednost posameznega registra na sklad
POP Reg32	Shrani trenutno vrednost na skladu v register
PUSHAD	Shrani vrednosti vseh splošno uporabnih registrov na sklad (vrstni red: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI)
POPAD	Shrani vrednosti sklada v registre
PUSHF	Shrani vrednost EFLAGS registra na sklad
POPF	Shrani trenutno vrednost sklada v EFLAGS register

Tabela 5.1: Ukazi za operacije s skladom in registri

Ti ukazi nam omogočajo, da shranimo vrednosti registrov na sklad, preden izvedemo odvečno kodo. Po izvedbi odvečne kode shranjene vrednosti povrnemo v registre in nadaljujemo z izvedbo programa. Med odvečno kodo se doda še kodo, ki onemogoča obratni zbirnik, ter tako še dodatno oteži samo analizo. Kot že omenjeno, se za onemogočanje obratnega zbirnika uporablja dva različna pristopa, ki sta podrobneje opisana v nadaljevanju.

5.1 Metoda s pogojnimi in nepogojnimi skoki

Pri tej metodi uporabimo kombinacijo pogojnih in nepogojnih skokov, ki onemogočijo obratni zbirnik pri analizi strojne kode. Tukaj je treba upoštevati pravilo, da se vedno izvede samo en skok in s tem ena veja programa. Tako onemogočimo analizo obratnemu zbirniku, saj le-ta analizira obe veji programa, kar je razvidno tudi iz slike 3.12, ki prikazuje delovanje rekurzivnega algoritma obratnega zbirnika.

Izpis 5.1 prikazuje odvečno kodo, ki uporablja kombinacijo JC in JMP ukaza. Na začetku shranimo EFLAGS register z ukazom PUSHF, nato s CLC pobrišemo prekoračitveni bit. Sledi JC pogojni skok, ki se izvede le v primeru, če je CF bit v EFLAGS registru postavljen na 1 (glej tabelo 2.7), kar pa v našem primeru ne bo nikoli, saj ga pobrišemo s CLC ukazom. Na koncu se

izvede JMP ukaz, ki preusmeri izvajanje programa na POPF ukaz, se pravi preskoči besedo 0xC0, ki je dejansko odvečna koda. Izpis 5.2 prikazuje napačno analizirano zbirno kodo obratnega zbirnika IDA.

```
__asm
{
    pushf //shranimo vrednost EFLAGS registra
    clc //nastavimo Carry bit na 0
    _emit 0x72 //
    _emit 0x02 // JC $ + 4; skok se ne bo nikoli izvedel
    _emit 0xEB
    _emit 0x01 // JMP $ + 3; skok se vedno izvede
    _emit 0xC0 // odvečna koda
    popf //povrnemo vrednosti EFLAGS registra
}
MessageBoxA(0,"Uporaba odvečne kode","",0);
```

Izpis 5.1: Uporaba pogojnega skoka za prekoračitev

```
.TEXT:00401000 PUSHFW
.TEXT:00401002 CLC
.TEXT:00401003 JB SHORT LOC_401007
.TEXT:00401005 JMP SHORT NEAR PTR LOC_401007+1
.TEXT:00401007 SHL BYTE PTR [ESI-63H], 6AH
.TEXT:0040100B ADD BYTE PTR (UNK_403FFF-40407BH)[EAX], CH
.TEXT:0040100E XOR EAX, [EAX+0]
.TEXT:00401011 PUSH OFFSET Text ; "Uporaba odvečne kode"
.TEXT:00401016 PUSH 0 ; hwnd
.TEXT:00401018 Call DS:__imp__MessageBoxA@16 ; MessageBoxA(x,x,x,x)
```

Izpis 5.2: Napačno analizirana koda obratnega zbirnika IDA

Za kontroliranje pogojnih skokov uporabimo EFLAGS register. Izpis 5.3 prikazuje odvečno kodo, ki uporablja EFLAGS register za kontrolo JC pogojnega skoka. Prvi štiri ukazi shranijo vrednost EFLAGS in EAX registra na sklad, vrednosti EAX registra pa se nato s XOR ukazom postavi na 0. Iz sklada se nato s POP ukazom v EAX register prekopira vrednosti EFLAGS registra. Iz tabele 2.7 vidimo, da je CF najmanj uteženi bit (angl. less significant bit), zato se za AND masko vzame binarno vrednosti 11111110 (šestnajstiška vrednost 0xFE). CF bit je tako postavljen na 0, vsi ostali biti pa ostanejo nespremenjeni. Vrednost se nato shrani na sklad in se jo s POPF ukazom prekopira v EFLAGS register. Iz sklada se nato povrne začetna vrednost v

EAX register, sledi še odvečna koda, ki deluje na že prej opisan način.

```
__asm
{
    pushf // shranimo vrednost EFLAGS na sklad
    push eax // shranimo vrednost EAX na sklad
    pushf // ponovno shranimo EFLAGS na sklad
    xor eax,eax // EAX = 0
    pop eax // vrednost EFLAGS damo iz sklada v EAX
    and eax,0FEh // CF = 0
    push eax
    popf // shranimo vrednost EAX v EFLAGS
    pop eax // povrnemo originalno vrednost v EAX
    _emit 0x72 //
    _emit 0x02 // JC $ + 4; skok se nebo nikoli izvedel
    _emit 0xEB
    _emit 0x01 // JMP $ + 3; skok se bo vedno izvedel
    _emit 0xC0 // odvečna koda
    popf //povrnemo vrednosti EFLAGS registra
}
MessageBoxA(0,"Uporaba odvečne kode","",0);
```

Izpis 5.3: Primer uporabe EFLAGS registra

5.2 Metoda s klicem podprograma

Ob klicu procedure se na vrhu sklada shrani naslov ukaza, ki se bo izvedel ko se bo končalo izvajanje podprograma. Ta naslov lahko znotraj podprograma spremenimo in tako kontroliramo sam potek programske kode. S tem pa tudi onemogočimo obratni zbirnik, saj bo le-ta analiziral kodo, ki sledi klicu podprograma, ki pa se ne bo nikoli izvedla. S to metodo onemogočimo obratnemu zbirniku tudi izdelavo referenc. Te se ustvarijo ob vejitvah ali klicu podprograma, obratni zbirnik pa z njimi analizira podprograme (število parametrov, tip parametrov, lokalne spremenljivke). Kot alternativo klicu podprograma lahko uporabimo kombinacijo PUSH in RET ukazov.

Izpis 5.4 prikazuje odvečno kodo, ki uporablja klic podprograma za onemogočanje obratnega zbirnika. Najprej se na sklad shrani vrednost EFLAGS registra, nato pa se preusmeri izvajanje programa s klicem podprograma na

ukaz PUSH EAX. Pri klicu podprograma se na sklad shrani povratni naslov, ki kaže na ukaz, ki se bo izvedel po končanem izvajanju podprograma, kar je v našem primeru XOR EAX, ESI ukaz. Iz sklada se nato v EAX register prenese ta naslov, ki se mu prišteje vrednost 18 (0x12 šestnajstiško). Vrednost je velikost vseh ukazov (v besedah) od XOR do POPF. Na sklad se nato prenese nov naslov, v EAX register se povrne stara vrednost, z RET ukazom pa se prekine izvajanje podprograma. Spremenjen povratni naslov kaže na POPF ukaz, od koder se tudi program izvaja naprej. Izpis 5.5 prikazuje napačno analizirano zbirno kodo. Iz izpisa tudi vidimo, da smo uspeli z odvečno kodo 0xC1 od ukaza XOR EAX, ESI onemogočiti obratni zbirnik, da bi pravilno analiziral kodo podprograma.

```
__asm
{
    pushf // shranimo EFLAGS register
    call $ + 8 // klic podprograma
    xor eax,esi //koda se nebo nikoli izvedla
    _emit 0xC1 //odvečna koda
    push eax // shranimo EAX vrednost (prvi ukaz podprograma)
    mov eax,[esp + 4] // v EAX damo naslov od PUSH EAX ukaza
    add eax,0x12 // spremenimo naslov da kaže na POPF ukaz
    mov [esp + 4],eax // shranimo nov naslov na sklad
    pop eax // povrnemo vrednost EAX registra
    ret // klic ki konča podprogram
    _emit 0xC1 //odvečna koda
    popf // ukaz ki se izvede po RET ukazu
}
MessageBoxA(0,"Uporaba odvečne kode","",0);
```

Izpis 5.4: Primer uporabe s klicem podprograma

```
.TEXT:00401000 PUSHFW
.TEXT:00401002 CALL LOC_401009 + 1
.TEXT:00401007 XOR EAX, ESI
.TEXT:00401009 LOC_401009:
.TEXT:00401009 RCL DWORD PTR [EAX−75H], 44H
.TEXT:0040100D AND AL, 4
.TEXT:0040100F ADD EAX, 12H
.TEXT:00401012 MOV [ESP+4], EAX
.TEXT:00401016 POP EAX
.TEXT:00401017 RETN
```

```
.TEXT:00401018 SHL DWORD PTR [ESI−63H], 6AH
.TEXT:0040101C ADD BYTE PTR (UNK_403FFF−40407BH)[EAX], CH
.TEXT:0040101F XOR EAX, [EAX+0]
.TEXT:00401022 PUSH OFFSET Text ; "Uporaba odvečne kode"
.TEXT:00401027 PUSH 0 ; hWnd
.TEXT:00401029 CALL DS:__imp__MessageBoxA@16 ; MessageBoxA(x,x,x,x)
.TEXT:0040102F RETN
```

Izpis 5.5: Napačno analizirana koda obratnega zbirnika IDA

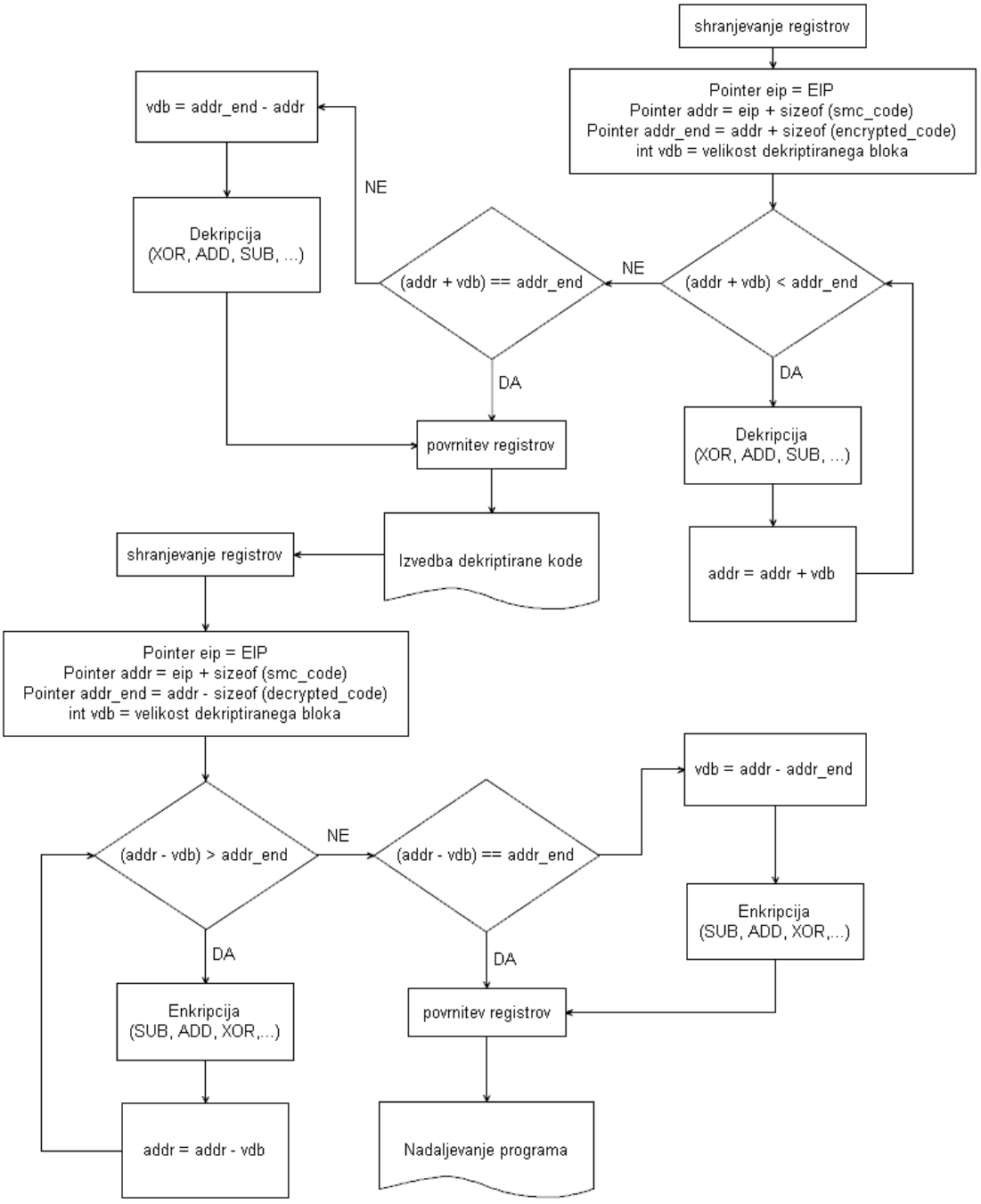
Poglavje 6

Samoreplikacijska koda

Samoreplikacijska koda spreminja strojno kodo med samim izvajanjem, navadno za zmanjšanje števila strojnih ukazov ali izboljšanje učinkovitosti strojne kode. Samorepliciranje dosežemo na različne načine, ki so odvisni od programskega jezika, ki ga uporabljamo. Načini za izgradnjo samoreplikacijske kode so naslednji:

- zamenjava obstoječega strojnega ukaza (ali dela ukaza, kot na primer: zamenjava registrov, naslova itd.),
- izdelava nadomestnih strojnih ukazov (glej 6.2) v pomnilniku,
- izdelava ali zamenjava izvorne kode, za kar potrebujemo dostop do sprotnega prevajalnika (primer je recimo ukaz *eval* v JavaScript-u),
- izdelava celotnega programa dinamično.

V diplomski nalogi smo se osredotočili na samoreplikacijsko kodo, ki bo kodirala in dekodirala programsko kodo. Za samo kodiranje in dekodiranje bi lahko uporabili keterkoli simetrični kriptografski algoritem (npr. Blowfish, Triple DES, Rijandel), vendar bi te rutine zelo povečale velikost ter zmanjšale hitrost izvajanja programske kode. Zato smo v samoreplikacijski kodi uporabili strojne ukaze kot so XOR, ADD, SUB ter še nekatere druge, ki se v procesorju hitro izvedejo. Prav tako moramo zagotoviti, da se bo koda izvajala kjer koli v programu, zato moramo poznati kazalec na ukaz, s katerim se v pomnilniku začne samoreplikacijska koda [24]. Slika 5.1 prikazuje uporabo odvečne kode kot prostor za samoreplikacijsko kodo, slika 6.1 pa prikazuje diagram neodvisne samoreplikacijske kode, ki smo jo uporabili v diplomski nalogi.



Slika 6.1: Diagram izvajanja neodvisne samoreplikacijske kode

Da neodvisna samoreplikacijska koda ne bo vplivala na samo delovanje programa, moramo na začetku najprej shraniti EFLAGS in vse ostale registre, ki se jih uporablja za dekodiranje strojne kode. Za dostop do kode moramo poznati naslov, na katerega kaže EIP register. Za branje in pisanje v register ne obstaja noben zbirni ukaz, spreminjamo ga s skočnimi ukazi ali klici podprograma. S klicem podprograma se na sklad shrani naslov naslednjega ukaza, ki se bo izvedel ob vrnitvi iz podprograma. Ta naslov uporabimo za relativno naslavljanje pri vseh operacijah samoreplikacijske kode. Izpis 6.1 prikazuje primer zbirne kode, ki v EDX register nastavi naslov, na katerem se nahaja POP EDX ukaz.

```
__asm
{
    push edx //shranimo vrednost EDX registra
    call $ + 5 // klic podprograma
    pop edx // EDX drži naslov od POP EDX ukaza
}
```

Izpis 6.1: Primer uporabe klica podprograma za pridobitev naslova v EIP registru

Za detekcijo bi lahko uporabili tudi krmilnik izjem (angl. exception handler), saj se ob generiranju izjeme shrani tudi EIP naslov, na katerem je do izjeme prišlo. Vendar bi za to potrebovali občutno več kode, kar pa ni optimalno za izvajanje samoreplikacijske kode. Ko poznamo naslov iz EIP registra, sledi dekodiranje zbirne kode. Pred izvedbo dekodiranega dela programa je potrebno povrniti vse prej shranjene registre. Po izvedbi dekodirane kode se spet shrani vse registre, ki se bodo uporabljali pri procesu kodiranja. Sledi pridobivanje naslova iz EIP registra, kodiranje strojne kode, povrnitev shranjenih registrov ter nadaljevanje izvajanja programa. Tako imamo v pomnilniku zmeraj dekodiran samo del strojne kode, ki se bo trenutno izvedel, vsi ostali deli pa so zakodirani.

Samoreplikacijska koda pogosto uporablja tudi tehniki polimorfizma in metamorfizma, ki jih bomo podrobneje opisali v nadaljenju.

6.1 Polimorfizem

Polimorfizem pri samoreplikacijski kodi omogoča, da se z vsako izvedbo koda drugače dekodira in zakodira, sintaksa programske kode pa ostane nespremenjena. To tehniko s pridom uporabljajo razne zlonamerne kode za skrivanje pred

protivirusnimi programi. Napredni polimorfični stroji vsebujejo tudi možnost generiranja odvečne kode, pogojnih skokov ter lažnih rutin (angl. dummy routines). Primer preproste polimorfične samoreplikacijske kode prikazuje izpis 6.2. Najprej shranimo vrednost EBX registra in pridobimo naslov, na katerega kaže EIP register. Nato uporabimo ključ 0x12345678 za dekodiranje ukazov PUSH EAX, PUSH EDX in RDTSC, ki se nato izvedejo. RDTSC ukaz vrne 64-bitno vrednost visoko ločljivostnega števca (glej 4.2.3) v EDX in EAX register. Vrednosti v EAX registru prištejemo vrednost ključa, nato ključ zamenjamo z vrednostjo v EAX registru. Sledi še kodiranje vrednost z novim ključem ter povrnitev registrov v prejšne stanje.

```
__asm{
    push ebx
    call $ + 5
    pop ebx // EBX drži EIP naslov od pop ebx ukaza
    xor dword ptr [ebx + 8],12345678h // za prvič se uporabi fiksen ključ
    _emit 0x28 // push eax
    _emit 0x04 // push edx
    _emit 0x3B // rdtsc
    _emit 0x23 //
    add eax,dword ptr [ebx + 4] // vrednost EAX registra prištejemo ključ
    mov dword ptr [ebx + 4],eax // nadomestimo ključ v kodi
    xor dword ptr [ebx + 8],eax // izvedemo enkripcijo z novim ključem
    pop edx //
    pop eax // obnovimo stare vrednosti registrov
    pop ebx //
}
```

Izpis 6.2: Primer polimorfične samoreplikacijske kode

Za prikaz delovanja polimorfične samoreplikacijske kode smo kodo iz izpisa 6.2 izvedli v zanki. Izpisi 6.3, 6.4, 6.5, 6.6 prikazujejo dva cikla izvajanja polimorfične samoreplikacijske kode. Iz izpisov lahko vidimo, da se polimorfična koda vedno zakodira z drugačnim ključem (izpisa 6.3 in 6.5), dekodirana koda pa je vedno enaka, kar se vidi iz izpisov 6.4 in 6.6.

```
00401007 PUSH EBX
00401008 CALL SmcTest.0040100D
0040100D POP EBX
0040100E XOR DWORD PTR DS:[EBX+8],12345678
00401015 SUB BYTE PTR DS:[EBX+EDI],AL
```

```
00401018 AND EAX,DWORD PTR DS:[EBX]
0040101A INC EBX
0040101B ADD AL,89
0040101D INC EBX
0040101E ADD AL,31
00401020 INC EBX
00401021 OR BYTE PTR DS:[EDX+58],BL
00401024 POP EBX
```

Izpis 6.3: Kodirana polimorfična samoreplikacijska koda (cikel 1)

```
00401007 PUSH EBX
00401008 CALL SmcTest.0040100D
0040100D POP EBX
0040100E XOR DWORD PTR DS:[EBX+8],12345678
00401015 PUSH EAX
00401016 PUSH EDX
00401017 RDTSC
00401019 ADD EAX,DWORD PTR DS:[EBX+4]
0040101C MOV DWORD PTR DS:[EBX+4],EAX
0040101F XOR DWORD PTR DS:[EBX+8],EAX
00401022 POP EDX
00401023 POP EAX
00401024 POP EBX
```

Izpis 6.4: Dekodirana polimorfična samoreplikacijska koda (cikel 1)

```
00401007 PUSH EBX
00401008 CALL SmcTest.0040100D
0040100D POP EBX
0040100E XOR DWORD PTR DS:[EBX+8],5B28F495
00401015 LDS ESP,FWORD PTR DS:[ESI+43036A27]
0040101B ADD AL,89
0040101D INC EBX
0040101E ADD AL,31
00401020 INC EBX
00401021 OR BYTE PTR DS:[EDX+58],BL
00401024 POP EBX
```

Izpis 6.5: Kodirana polimorfična samoreplikacijska koda (cikel 2)

```
00401007 PUSH EBX
00401008 CALL SmcTest.0040100D
0040100D POP EBX
0040100E XOR DWORD PTR DS:[EBX+8],5B28F495
00401015 PUSH EAX
00401016 PUSH EDX
00401017 RDTSC
00401019 ADD EAX,DWORD PTR DS:[EBX+4]
0040101C MOV DWORD PTR DS:[EBX+4],EAX
0040101F XOR DWORD PTR DS:[EBX+8],EAX
00401022 POP EDX
00401023 POP EAX
00401024 POP EBX
```

Izpis 6.6: Dekodirana polimorfična samoreplikacijska koda (cikel 2)

6.2 Metamorfizem

Metamorfizem je tehnika, kjer programska koda spreminja svojo sintakso, ne spremeni pa semantike programske kode. Koda lahko spreminja izvorne in ponorne registre, vstavlja NOP ukaze ter nepogojne skoke. Nekateri metamorfični stroji celo omogočajo generiranje nadomestnih ukazov za isto operacijo. Izpisi 6.7, 6.8 in 6.9 prikazujejo operacijo, ki register EAX postavi na vrednost 0x12345678. Izpisi 6.7 prikazuje MOV ukaz, ki EAX register postavi na vrednost 0x12345678.

```
__asm{
    mov eax, 0x12345678
}
```

Izpis 6.7: Operacija z MOV ukazom

Izpis 6.8 prikazuje operacijo, ki uporablja MOV in XOR ukaz za nastavljanje registra. Najprej se vrednost EAX registra z MOV ukazom nastavi na 0x9257272D, potem pa se nad to vrednostjo izvede logično operacijo XOR z vrednostjo 0x80637155. Po operaciji ima EAX register vrednost 0x12345678.

```
__asm{
    mov eax, 0x9257272D
```

```
        xor eax, 0x80637155
    }
```

Izpis 6.8: Operacija z MOV in XOR ukazoma

Izpis 6.9 prikazuje kombinacijo MOV in LEA ukazov. Najprej se z MOV ukazom EBX register nastavi na vrednost 0x091A2B3C. Nato LEA ukaz pomnoži vrednost EBX registra z 2, rezultat pa shrani v EAX register. Po izvedbi LEA ukaza je EAX register nastavljen na vrednost 0x12345678.

```
    __asm{
        mov ebx, 0x091A2B3C
        lea eax, [ebx*2]
    }
```

Izpis 6.9: Operacija z MOV in LEA ukazoma

Za tako delovanje mora metamorfični stroj najprej interpretirati strojno kodo. Interpretacijo nato spremeni ter pretvori v strojno kodo. Tak metamorfičen stroj je sestavljen iz vsaj treh delov, obratnega zbirnika, permutacijske rutine ter zbirnika.

Obratni zbirnik:
Obratni zbirnik ima funkcijo, da pretvori strojni jezik v notranjo interpretacijo strojnega jezika. Ta jezik se lahko jemlje kot nekakšen interni jezik, s katerim zna delati samo metamorfični stroj.

Permutacijska rutina:
Permutacijska rutina nato spremeni potek strojnih ukazov, zamenja uporabljene registre, lahko pa določeno zaporedje ukazov spremeni z zaporedjem drugih ukazov, kot je razvidno iz izpisov 6.7, 6.8 in 6.9. Vstavi tudi odvečno kodo ali razne nepogojne skoke, kar dodatno spremeni sam potek izvedbe programske kode.

Zbirnik:
Zbirnik nato interpretacijo originalnega strojnega jezika spremeni nazaj v strojni jezik. Novo generirana koda je tako sintaktično različna kot prejšna, vendar je rezultat izvedbe popolnoma enak.

Za razvoj metamorfičnega stroja, ki bi znal določeno zaporedje ukazov zamenjati z nadomestnimi, bi potrebovali kar nekaj časa, zato smo za primer metamorfične kode razvili preprosto zbirno kodo, ki pridobi vrednost registra EIP, in je prikazana v izpisu 6.10.

```
__asm{
    push eax
    push edx
    rdtsc
    push edx // 1. sprememba
    call $ + 5
    pop edx // 2. sprememba
    pushad //shrani vrednost vseh registrov na sklad
    and eax,0x00000003 //EAX vsebuje prve tri bite RDTSC vrednosti
    cmp eax,0
    _emit 0x74
    _emit 0x0C //JZ $ + 14
    cmp eax,4
    _emit 0x74
    _emit 0x07 //JZ $ + 9
    cmp eax,5
    _emit 0x74
    _emit 0x02 //JZ $ + 4
    jmp short $ + 5
    or eax,2
    mov ecx,edx //3. sprememba
    and byte ptr [ecx − 6], 0xF8 // zadnji trije biti postavljeni na 0
    or byte ptr [ecx − 6], al // nova vrednost zadnjih treh bitov
    and byte ptr [ecx], 0xF8 // zadnji trije biti postavljeni na 0
    or byte ptr [ecx], al // nova vrednost zadnjih treh bitov
    and byte ptr [ecx + 0x1A],0xF8 // zadnji trije biti postavljeni na 0
    or byte ptr [ecx + 0x1A], al
    popad
    add esp,4 //popravek sklada
    pop edx
    pop eax
}
```

Izpis 6.10: Primer metamorfične kode

Iz izpisa je razvidno, da se najprej shranita vrednosti EAX in EDX registrov na sklad, nato se kliče RDTSC ukaz, ki v ta dva registra shrani vrednost 64-bitnega visokoločljivostnega števca. Nato se shrani vrednost EDX registra, kliče podprogram ter prenese vrednost EIP registra z vrha sklada v EDX register. Sledi preprosta metamorfična rutina, ki spremeni ponorni register, v katerega se bo shranila vrednost EIP registra. Pri pisanju metamorfične ru-

tine smo si pomagali s tabelo 6.1, ki prikazuje vse kombinacije PUSH in POP zbirnega ukaza za vse splošno dostopne registre.

Ukaz	Op. koda	Binarna vrednost	Ukaz	Op. koda	Binarna vrednost
push eax	0x50	01010000	pop eax	0x58	01011000
push ebx	0x53	01010011	pop ebx	0x5B	01011011
push ecx	0x51	01010001	pop ecx	0x59	01011001
push edx	0x52	01010010	pop edx	0x5A	01011010
push esi	0x56	01010110	pop esi	0x5E	01011110
push edi	0x57	01010111	pop edi	0x5F	01011111
push esp	0x54	01010100	pop esp	0x5C	01011100
push ebp	0x55	01010101	pop ebp	0x5D	01011101

Tabela 6.1: Kombinacije PUSH in POP ukaza za vse splošno dostopne registre

Iz tabele vidimo, da se vse kombinacije PUSH in POP ukazov razlikujejo le v prvih treh bitih, ki določajo izvorni ali ponorni register (glej tabelo 2.11). Metamorfična rutina v našem primeru ne sme generirati vseh osem kombinacij registrov, saj se register EAX uporablja za pridobivanje prvih 32 bitov visokoločljivostnega števca, EBP in ESP pa kažeta na sklad, v katerega shranjujemo trenutne vrednosti registrov. Tako nam ostane kombinacija petih registrov, ki jih lahko uporabimo za pridobivanje vrednosti EIP registra. Metamorfična rutina najprej shrani vrednosti vseh registrov z ukazom PUSHAD, nato z operacijo AND postavi vse bite na 0, razen prvih treh, ki se bodo uporabili za nastavljanje vrednosti za izvorni ali ponorni register. Sledi primerjanje vrednosti v EAX registru z vrednostmi, ki bi nam za izvorni ali ponorni register dali registre EAX, EBP ali ESP (glej tabelo 2.11). Če se taka vrednost nahaja v EAX registru, rutina nastavi drugi bit te vrednosti na 1, ter s tem zagotovi, da se napačna vrednost nikoli ne bo uporabila. Sledi premik vrednosti iz EDX registra v ECX register, nato se z AND in OR operacijami nastavi nove vrednosti za izvorne ali ponorne registre v ukazih, ki delajo z vrednostjo EIP registra. Sledi še obnovitev starih registrov ter popravek sklada. Metamorfično kodo smo za testiranje poganjali v zanki, rezultate pa prikazujeta izpisa 6.11 in 6.12.

```
00401026 PUSH EAX
00401027 PUSH EDX
```

```
00401028 RDTSC
0040102A PUSH ECX
0040102B CALL SmcTest.00401030
00401030 POP ECX
00401031 PUSHAD
00401032 AND EAX,3
00401035 CMP EAX,0
00401038 JE SHORT SmcTest.00401046
0040103A CMP EAX,4
0040103D JE SHORT SmcTest.00401046
0040103F CMP EAX,5
00401042 JE SHORT SmcTest.00401046
00401044 JMP SHORT SmcTest.00401049
00401046 OR EAX,2
00401049 MOV ECX,ECX
0040104B AND BYTE PTR DS:[ECX-6],0F8
0040104F OR BYTE PTR DS:[ECX-6],AL
00401052 AND BYTE PTR DS:[ECX],0F8
00401055 OR BYTE PTR DS:[ECX],AL
00401057 AND BYTE PTR DS:[ECX+1A],0F8
0040105B OR BYTE PTR DS:[ECX+1A],AL
0040105E POPAD
0040105F ADD ESP,4
00401062 POP EDX
00401063 POP EAX
```

Izpis 6.11: Metamorfična koda po prvi izvedbi

```
00401026 PUSH EAX
00401027 PUSH EDX
00401028 RDTSC
0040102A PUSH EBX
0040102B CALL SmcTest.00401030
00401030 POP EBX
00401031 PUSHAD
00401032 AND EAX,3
00401035 CMP EAX,0
00401038 JE SHORT SmcTest.00401046
0040103A CMP EAX,4
0040103D JE SHORT SmcTest.00401046
0040103F CMP EAX,5
00401042 JE SHORT SmcTest.00401046
```

```
00401044 JMP SHORT SmcTest.00401049
00401046 OR EAX,2
00401049 MOV ECX,EBX
0040104B AND BYTE PTR DS:[ECX−6],0F8
0040104F OR BYTE PTR DS:[ECX−6],AL
00401052 AND BYTE PTR DS:[ECX],0F8
00401055 OR BYTE PTR DS:[ECX],AL
00401057 AND BYTE PTR DS:[ECX+1A],0F8
0040105B OR BYTE PTR DS:[ECX+1A],AL
0040105E POPAD
0040105F ADD ESP,4
00401062 POP EDX
00401063 POP EAX
```

Izpis 6.12: Metamorfična koda po tretji izvedbi

Iz obeh izpisov vidimo, da se na naslovih 0x0040102A, 0x00401030 in 0x00401049 spreminjajo registri, ki jih ukazi uporabljajo. V izpisu 6.11 na naslovu 0x00401049 celo vidimo, da je metamorfična rutina zgenerirala ukaz MOV ECX, ECX, ki je popolnoma neuporaben, saj vrednost ECX registra premakne v ECX register. Vendar pa je za metamorfično rutino ukaz logičen, saj se za pridobivanje vrednosti EIP registra v tem ciklu uporablja ECX register.

Poglavje 7

Primer uporabe

V diplomski nalogi smo predstavili različna orodja za izvajanje obratnega inženiringa ter metode, ki to delo otežijo. Na spletu obstaja kar nekaj programske opreme, kot so kodirniki in pakirniki (glej podpoglavje 4.4), ki se uporablja za zaščito pred obratnim inženiringom. Večina uporablja v diplomi naštete metode zaznavanja razhroščevalnikov in odvečno kodo za onemogočanje obratnega zbirnika. S temi metodami skušajo zakriti svoje metode za odpakiranje ali dekodiranje programa v pomnilnik, saj se odpakiran program iz pomnilnika da prenesti na disk ter tako pridobiti strojno kodo. Tako smo se za diplomsko nalogo odločili, da bomo razvili orodje, ki bo za zaščito programske kode uporabljalo tehniki odvečne in samoreplikacijske kode. Sestavljeno je iz dveh delov:

- razvojno programski paket (angl. software development kit ali SDK),
- generator samoreplikacijske kode.

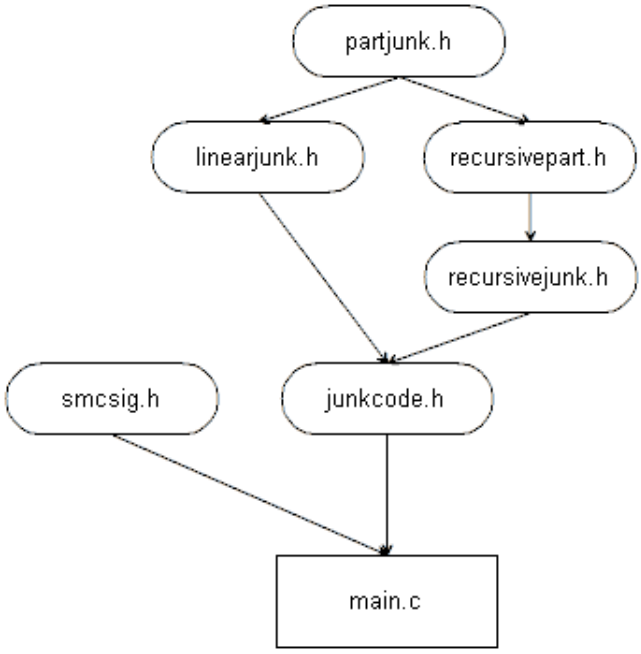
Orodje omogoča programerju, da že med samim razvojem programa označi predele programske kode, ki se bodo posebej kodirali s samoreplikacijsko kodo. Prav tako vsebuje makroje, ki med programsko kodo vstavijo odvečno kodo, ki onemogoči obratni zbirnik. Vse to omogoča programerju, da še dodatno oteži samo izvajanje obratnega inženiringa nad kodo programa.

7.1 Razvojno-programski paket

Paket je skupek zaglavnih datotek (angl. header files), v katerih je napisana odvečna koda v obliki makrojev. Odvečno kodo smo razdelili na dva dela:

- koda, ki označuje začetek in konec programskega bloka, ki se bo kodiral,
- koda, ki onemogoča obratni zbirnik.

Slika 7.1 prikazuje relacijo zaglavnih datotek v razvojno-programskem paketu. Osnovna datoteka je partjunk.h, ki vsebuje osnovne makroje za generiranje odvečne kode. Datoteki linearjunk.h in recursivepart.h sta izpeljani iz prej omenjene datoteke, prva vsebuje makroje za onemogočanje linearnega algoritma, druga pa razširjene makroje, ki so sestavni del makrojev v datoteki recursivejunk.h. Ti makroji onemogočajo rekurzivni algoritem obratnega zbirnika. Sledi še datoteka junkcode.h, ki je samo ovojnica za prej naštete datoteke, tako da se v projekt vključi samo datoteko junkcode.h. Datoteka smcsig.h pa vsebuje samo dva makroja odvečne kode, ki označujeta začetek in konec bloka, ki se bo kodiral s samoreplikacijsko kodo.



Slika 7.1: Relacija zaglavnih datotek razvojno-programskega paketa

Za tak koncept smo se odločili, ker je tako veliko lažje med seboj kombinirati različne makroje ter generirati različno odvečno kodo. Nekatere makroje smo razširili s parametri in omogočili, da lahko programer še dodatno nastavlja obnašanje odvečne kode, ne da bi za to moral napisati svoj makro. Izpis 7.1 prikazuje enega izmet takih makrojev.

```
#define RECURSIVE_JUNK_CALL_RET_JMP(X,Y) \
        PARTIAL_JUNK_PUSH_EAX \
        PARTIAL_JUNK_CALL_X(0x04) \
        PARTIAL_JNZ(X) \
        PARTIAL_JMP(Y) \
        PARTIAL_JUNK_POP_EAX \
        PARTIAL_JMP(0x02) \
        __asm _emit 0xEB \
        __asm _emit 0x06 \
        PARTIAL_ADD_EAX_X(0x07) \
        PARTIAL_JUNK_PUSH_EAX \
        __asm _emit 0xC3 \
        __asm _emit 0xE8 \
        PARTIAL_JUNK_POP_EAX
```

Izpis 7.1: Primer makroja s parametri

Makro uporablja parametra X in Y kot parametra za JNZ in JMP ukaza, ki se v odvečni kodi nikoli ne izvedeta, vendar pa veliko pripomoreta za onemogočanje obratnega zbirnika. Izpisa 7.2 in 7.3 prikazujeta uporabo makroja s slabimi parametri in analizo obratnega zbirnika take kode.

```
#include <windows.h>
#include "junkcode.h"

int main(int argc, char *argv[]){
    RECURSIVE_JUNK_CALL_RET_JMP(0x01,0x01)
    MessageBoxA(0,(LPCSTR)"test",(LPCSTR)"test",0);
}
```

Izpis 7.2: Primer uporabe makroja s slabimi parametri

```
.TEXT:00401000 PUSH EAX
.TEXT:00401001 CALL LOC_40100A
.TEXT:00401006 JNZ SHORT NEAR PTR LOC_401008+1
.TEXT:00401008 JMP SHORT LOC_40100B
.TEXT:0040100A LOC_40100A:
.TEXT:0040100A POP EAX
.TEXT:0040100B LOC_40100B:
.TEXT:0040100B JMP SHORT LOC_40100F
.TEXT:0040100D JMP SHORT LOC_401015
```

```
.TEXT:0040100F LOC_40100F:
.TEXT:0040100F ADD EAX, 7
.TEXT:00401012 PUSH EAX
.TEXT:00401013 RETN
.TEXT:00401014 DB 0E8H
.TEXT:00401015 LOC_401015:
.TEXT:00401015 POP EAX
.TEXT:00401016 PUSH 0 ; uType
.TEXT:00401018 PUSH OFFSET Caption ; "test"
.TEXT:0040101D PUSH OFFSET Text ; "test"
.TEXT:00401022 PUSH 0 ; hWnd
.TEXT:00401024 CALL DS:__imp__MessageBoxA@16 ; MessageBoxA(x,x,x,x)
.TEXT:0040102A RETN
```

Izpis 7.3: Primer analize zbirne kode s slabimi parametri makroja

Iz izpisa 7.3 vidimo, da se odvečna koda sicer nahaja pred API klicem MessageBoxA, vendar ne vpliva na samo analizo obratnega zbirnika. Problem je v slabo nastavljenih parametrih, ki se uporabljajo pri klicih JNZ in JMP na naslovih 0x00401006 in 0x00401008. Če popravimo parametre makroja, kot prikazuje izpis 7.4, potem odvečna koda v večji meri vpliva na analizo obratnega zbirnika, ki je prikazana v izpisu 7.5.

```
#include <windows.h>
#include "junkcode.h"

int main(int argc, char *argv[]){
    RECURSIVE_JUNK_CALL_RET_JMP(0x11,0x15)
    MessageBoxA(0,(LPCSTR)"test",(LPCSTR)"test",0);
}
```

Izpis 7.4: Primer uporabe makroja z izboljšanimi parametri

```
.TEXT:00401000 PUSH EAX
.TEXT:00401001 CALL LOC_40100A
.TEXT:00401006 JNZ SHORT LOC_401019
.TEXT:00401008 JMP SHORT LOC_40101F
.TEXT:0040100A LOC_40100A:
.TEXT:0040100A POP EAX
.TEXT:0040100B JMP SHORT LOC_40100F
.TEXT:0040100D DB 0EBH, 6
.TEXT:0040100F LOC_40100F:
```

```
.TEXT:0040100F ADD EAX, 7
.TEXT:00401012 PUSH EAX ; LPTEXT
.TEXT:00401013 RETN
.TEXT:00401014 DD 6A58E8H
.TEXT:00401018 DB 68H
.TEXT:00401019 LOC_401019:
.TEXT:00401019 SBB [EAX+0], DH
.TEXT:0040101B INC EAX
.TEXT:0040101C ADD [EAX+20H], CH
.TEXT:0040101F LOC_40101F:
.TEXT:0040101F XOR [EAX+0], AL
.TEXT:00401022 PUSH 0 ; hWnd
.TEXT:00401024 CALL DS:__imp__MessageBoxA@16 ; MessageBoxA(x,x,x,x)
.TEXT:0040102A RETN
```

Izpis 7.5: Primer analize zbirne kode z izboljšanimi parametri makroja

Pri makrojih, ki se uporabljajo za označevanje blokov, ki se bodo kodirali s samoreplikacijsko kodo, velja omeniti še, da se ne smejo uporabljati v gnezdeni obliki, saj generator samoreplikacijske kode ne podpira gnezdenja. Izpisa 7.6 in 7.7 prikazujeta pravilno in nepravilno uporabo makroja za generiranje samoreplikacijske kode.

```
int main(int argc, char *argv[]){
    int i = 0;

    SMC_SIG_START //makro označuje začetek kodiranega bloka

    for(i = 0; i < 10; i++){
        printf("%d\n",i);
    }

    SMC_SIG_END //označuje konec kodiranega bloka
    return 0;
}
```

Izpis 7.6: Primer pravilne uporabe SDK-ja

```
int main(int argc, char *argv[]){
    int i = 0;

    SMC_SIG_START

    for(i = 0; i < 10; i++){
        SMC_SIG_START
        printf("%d\n",i);
        SMC_SIG_END
    }

    SMC_SIG_END
    return 0;
}
```

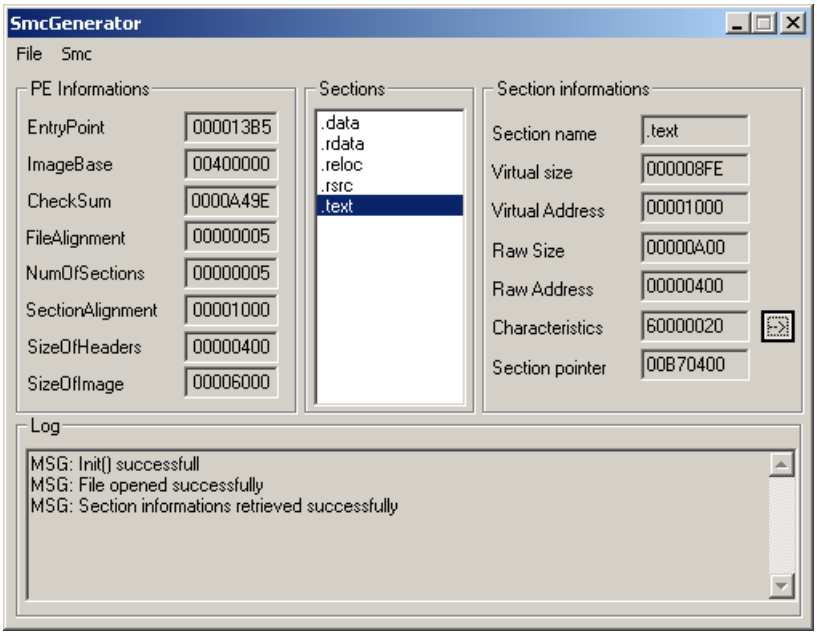
Izpis 7.7: Primer nepravilne uporabe SDK-ja zaradi gnezdenega označevanja programskih blokov

7.2 Generator samoreplikacijske kode

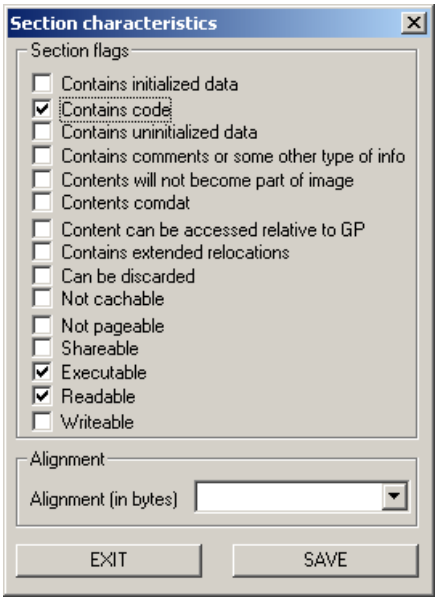
Generator samoreplikacijske kode je sestavljen iz dveh delov, odprtokodne dinamične knjižnice PELibrary 0.3 beta[1] ter grafičnega vmesnika. Knjižnica vsebuje API klice za delo s PE formatom, napisana je v zbirnem jeziku MASM. Grafični vmesnik aplikacije je napisan v programskem jeziku C, ves razvoj pa je potekal v razvojnem okolju Visual C++ 2008 Express Edition[2]. Grafični vmesnik je prikazan na sliki 7.2, vsebuje informacije o glavi PE, sekcijah ter komponento za zapisovanje informacij. Poleg polja Characteristics se nahaja tudi gumb, ki odpre dodaten dialog, v katerem lahko nastavljamo karakteristike za izbrano sekcijo. Dialog je prikazan na sliki 7.3.

[1] `http://sourceforge.net/projects/pe-lib/files/PeLibrary/PeLibrary-0.3b.zip/`

[2] `http://www.microsoft.com/visualstudio/en-us/products/2008-editions/express`



Slika 7.2: Grafični vmesnik generatorja samoreplikacijske kode



Slika 7.3: Dialog za nastavitev karakteristik sekcije

Delovanje generatorja samoreplikacijske kode je precej preprosto. Najprej se preko knjižnice PELibrary 0.3 naloži željen program v pomnilnik. Knjižnica

analizira PE format, rezultate pa posreduje generatorju, ki jih nato prikaže v grafičnem vmesniku. Uporabnik nato izbere sekcijo, nad katero se bo izvedlo iskanje odvečne kode, ki označuje programske bloke, ki se bodo kodirali. Iz informacij o blokih se zgradi dvojno-povezani seznam, vozlišče seznama prikazuje izpis 7.8.

```
typedef struct SMC_SIGNATURE_INFO
{
    DWORD StartAddress;
    DWORD VStartAddress;
    DWORD SigLength;
    DWORD SigType;
    void *nextSignature;
    void *prevSignature;
}
SMC_SIGNATURE_INFO, *pSMC_SIGNATURE_INFO;
```

Izpis 7.8: Vozlišče dvojno-povezanega seznama

Seznam se gradi zaradi lažjega povezovanja programskih blokov, ki so označeni za kodiranje s samoreplikacijsko kodo. Generator mora namreč poznati vrstni red programskih blokov za kodiranje, zato je najlažje to prikazati prav z dvojno-povezanim seznamom. Seznam se uporablja tudi za preverjanje gnezdenih makrojev, ker trenutno niso podprti. Ko je seznam zgrajen in preverjen, se začne kodiranje kode. Generator iz prvega vozlišča prebere vrednost polja SigType, ki je postavljeno na 0, kar označuje začetek kodiranega bloka. Nato se prebereta vrednosti StartAddress in SigLength, ki označujeta začetni pomnilniški naslov odvečne kode in njeno velikost. Iz teh podatkov se nato izračuna začetni naslov programske kode, ki se bo kodirala po naslednji formuli:

$$\text{StartOfCodeBlock} = \text{StartAddress} + \text{SigLength}$$

Polje nextSignature vsebuje kazalec na naslednje vozlišče, ki ima polje SigType postavljeno na 1 in označuje konec kodirnega bloka. Iz tega vozlišča se prebere vrednost polja StartAddress, ki označuje pomnilniški naslov, kjer se konča kodiran blok in začne odvečna koda. Ta naslov se uporabi za izračun velikost kodiranega bloka v besedah po naslednji formuli:

$$\text{BlockSize} = \text{StartAddress} - \text{StartOfCodeBlock}$$

Pridobljene podatke se uporabi za nastavitev parametrov samoreplikacijske kode, kot sta velikost kodirnega bloka in naključno generirana vrednost za ključ. Nato se rezervira dovolj velik blok pomnilnika, v katerega se skopira kodirni del samoreplikacijske kode, in kodo, ki se bo kodirala. Sledi izvajanje kodirne samoreplikacijske kode v pomnilniku. V primeru, da je prišlo do izjeme, se izjema zabeleži, izvajanje pa se prekine. V nasprotnem primeru se originalni blok kode zamenja s kodiranim, odvečno kodo, ki označuje začetek in konec kodiranega bloka, pa se zamenja z dekodirno in kodirno rutino. Na koncu je potrebno nastaviti še pravice pisanja za izbrano sekcijo (slika 7.3), saj bo drugače prišlo do izjeme pri izvajanju samoreplikacijske kode. Konkreten primer je opisan v naslednjem podpoglavju.

7.3 Rezultati

Za testiranje orodja smo napisali kratek program v C-ju, ki ga prikazuje izpis 7.11. Za pravilno geslo velja, da sta enaka prvi in tretji ter peti in šesti znak. Različna pa morata biti drugi in četrti ter sedmi in osmi znak.

```
#include <stdio.h>

int main(int argc, char *argv[]){
    int i = 0;
    char buffer[9];

    memset(buffer,0,9);
    printf("Vpisite geslo (najvec 8 znakov): ");
    fgets(buffer,9,stdin);

    /* Primer pravilnega gesla: 01023350 */
    if(buffer[0] == buffer[2] && buffer[1] != buffer[3] &&
        buffer[4] == buffer[5] && buffer[6] != buffer[7]){
        printf("Geslo je pravilno\n");
    }
    else{
        printf("Geslo je nepravilno");
    }
    return 0;
}
```

Izpis 7.9: Izvorna koda testnega programa

Za analizo strojne kode smo uporabili IDA Free obratni zbirnik. Izpis 7.10 prikazuje zbirno kodo, ki jo je izdela IDA. Kot vidimo iz izpisa, se za primerjanje uporabljajo registri CL, DL in AL. Najprej se znake kopira v registre, nato pa se vrednost registrov primerja s posameznimi vrednostmi v pomnilniku.

```
00401035 PUSH 9 ; MaxCount
00401037 PUSH EAX ; Buf
00401038 CALL DS:__imp__fgets
0040103E MOV CL, [ESP+24H+buffer]
00401042 ADD ESP, 10H
00401045 CMP CL, [ESP+14H+buffer+2]
00401049 JNZ SHORT 00401070
0040104B MOV DL, [ESP+14H+buffer+1]
0040104F CMP DL, [ESP+14H+buffer+3]
00401053 JZ SHORT 00401070
00401055 MOV AL, [ESP+14H+buffer+4]
00401059 CMP AL, [ESP+14H+buffer+5]
0040105D JNZ SHORT 00401070
0040105F MOV CL, [ESP+14H+buffer+6]
00401063 CMP CL, [ESP+14H+buffer+7]
00401067 JZ SHORT 00401070
00401069 PUSH OFFSET aGesloJePraviln ; "Geslo je pravilno\n"
0040106E JMP SHORT 00401075
00401070 PUSH OFFSET aGesloJeNepravi ; "Geslo je nepravilno"
00401075 CALL ESI ; __imp__printf
```

Izpis 7.10: Analiza zbirne kode testnega programa

Iz izpisa vidimo, da se da precej preprosto razbrati potek samega algoritma in ugotoviti veljavno geslo. Zato smo za naslednji test v izvorno kodo vključili makroje z odvečno kodo, kar je prikazano v izpisu 7.11.

```
#include <stdio.h>
#include "junkcode.h"

int main(int argc, char *argv[]){
    int i = 0;
    char buffer[9];

    memset(buffer,0,9);
    printf("Vpisite geslo (najvec 8 znakov): ");
```

```
    RECURSIVE_JUNK_CALL_RET_ECX
    fgets(buffer,9,stdin);

    /* 01023350 */
    RECURSIVE_JUNK_CALL_RET_EBX
    if(buffer[0] == buffer[2] && buffer[1] != buffer[3] &&
        buffer[4] == buffer[5] && buffer[6] != buffer[7]){
        RECURSIVE_JUNK_CALL_RET_EAX
        printf("Geslo je pravilno\n");
    }
    else{
        RECURSIVE_JUNK_CALL_RET_EDX
        printf("Geslo je nepravilno");
    }
    return 0;
}
```

Izpis 7.11: Izvorna koda testnega programa, opremljenega z makroji odvečne kode

Izpis 7.12 prikazuje analizo zbirne kode, ki vsebuje makroje za onemogočanje obratnega zbirnika. Iz izpisa je razvidno, da odvečna koda uspešno onemogoči analizo obratnega zbirnika. Iz izpisa celo vidimo, da obratni zbirnik ni analiziral strojne kode med naslovom 0x00401060 in 0x00401098, kjer se nahaja algoritem za preverjanje vpisanega gesla. Zbirna koda sicer vsebuje reference na nize (naslova 0x00401099 in 0x004010B0), vendar referenc nikakor ne moremo povezati s funkcijo printf(), saj referenca na to funkcijo v analizi zbirne kode ne obstaja.

```
.TEXT:00401034 LOC_401034:
.TEXT:00401034 POP ECX
.TEXT:00401035 ADD ECX, 8
.TEXT:00401038 PUSH ECX
.TEXT:00401039 RETN
.TEXT:0040103A DW 59E8H
.TEXT:0040103C CALL DS:__imp____iob_func
.TEXT:00401042 PUSH EAX ; File
.TEXT:00401043 LEA EAX, [ESP+1CH+Buf]
.TEXT:00401047 PUSH 9 ; MaxCount
.TEXT:00401049 PUSH EAX ; Buf
.TEXT:0040104A CALL DS:__imp__fgets
.TEXT:00401050 ADD ESP, 0CH
```

```
.TEXT:00401053 PUSH EBX
.TEXT:00401054 CALL LOC_40105A
.TEXT:00401059 DB 0E8H
.TEXT:0040105A LOC_40105A:
.TEXT:0040105A POP EBX
.TEXT:0040105B ADD EBX, 8
.TEXT:0040105E PUSH EBX
.TEXT:0040105F RETN
.TEXT:00401060 DD 4C8A5BE8H, 4C3A0424H, 34750624H, 524548AH
.TEXT:00401060 DD 724543AH
.TEXT:00401060 DD 448A2A74H, 443A0824H, 20750924H, 0A244C8AH
.TEXT:00401060 DD 0B244C3AH
.TEXT:00401060 DD 0E8501674H, 1, 0C08358E8H, 0E8C35008H
.TEXT:00401098 POP EAX
.TEXT:00401099 PUSH OFFSET aGesloJePraviln ; "Geslo je pravilno\n"
.TEXT:0040109E JMP SHORT LOC_4010B4
.TEXT:004010A0 PUSH EDX
.TEXT:004010A1 CALL LOC_4010A7
.TEXT:004010A6 DB 0E8H
.TEXT:004010A7 LOC_4010A7:
.TEXT:004010A7 POP EDX
.TEXT:004010A8 ADD EDX, 8
.TEXT:004010AB PUSH EDX
.TEXT:004010AC RETN
.TEXT:004010AD DB 0E8H, 5AH, 68H
.TEXT:004010B0 DD OFFSET aGesloJeNepravi ; "Geslo je nepravilno"
.TEXT:004010B4 LOC_4010B4:
.TEXT:004010B4 CALL ESI
.TEXT:004010B6 MOV ECX, DWORD PTR [ESP+1CH+buffer+8]
.TEXT:004010BA ADD ESP, 4
.TEXT:004010BD POP ESI
.TEXT:004010BE XOR ECX, ESP ; cookie
.TEXT:004010C0 XOR EAX, EAX
.TEXT:004010C2 CALL @__security_check_cookie@4 ; __security_check_cookie(x)
.TEXT:004010C7 ADD ESP, 10H
.TEXT:004010CA RETN
```

Izpis 7.12: Analiza zbirne kode z makroji označenega testnega programa

Za naslednji test smo poleg makrojev odvečne kode uporabili še makroje, ki označujejo programski blok, ki se bo kodiral s samoreplikacijsko kodo. Izvorno kodo prikazuje izpis 7.13. Za kodiranje smo označili programski blok, ki

obsega if/else stavek in kodo znotraj njega. V tem primeru smo makro RECURSIVE_JUNK_CALL_RET_EBX uporabili za zakritje dekodirne rutine, ki bo nadomestila odvečno kodo, ki se nahaja v makroju SMC_SIG_START.

```
#include <stdio.h>
#include "junkcode.h"
#include "smcsig.h"

int main(int argc, char *argv[]){
    int i = 0;
    char buffer[9];

    memset(buffer,0,9);
    printf("Vpisite geslo (najvec 8 znakov): ");
    RECURSIVE_JUNK_CALL_RET_ECX
    fgets(buffer,9,stdin);

    /* 01023350 */
    RECURSIVE_JUNK_CALL_RET_EBX
    SMC_SIG_START //začetek kodiranega bloka
    if(buffer[0] == buffer[2] && buffer[1] != buffer[3] &&
        buffer[4] == buffer[5] && buffer[6] != buffer[7]){
        RECURSIVE_JUNK_CALL_RET_EAX
        printf("Geslo je pravilno\n");
    }
    else{
        RECURSIVE_JUNK_CALL_RET_EDX
        printf("Geslo je nepravilno");
    }
    SMC_SIG_END //konec kodiranega bloka
    return 0;
}
```

Izpis 7.13: Izvorna koda testnega programa, opremljenega z makroji odvečne kode in makroji, ki označujejo začetek in konec kodiranega bloka

Izpis 7.14 prikazuje analizo zbirne kode, ki je kodirana s samoreplikacijsko kodo. Iz izpisa vidimo, da koda najprej kliče printf() funkcijo za izpis niza na standardni izhod (naslov 0x00401028), nato se začne odvečna koda, ki jo vsebuje makro RECURSIVE_JUNK_CALL_RET_EBX (naslov 0x0040102D). Ta onemogoči analizo zbirne kode od naslova 0x0040103A naprej in s tem zakrije dekodirno metodo samoreplikacijske kode, ki se začne prav na tem

naslovu. Zbirna koda je napačna do naslova 0x004010DD, kjer se začne kodirna rutina samoreplikacijske kode. Za zaščito kodirne rutine bi lahko uporabili katerega od prej navedenih makrojev, ki vsebuje odvečno kodo.

```
00401000 SUB ESP, 10H
00401003 MOV EAX, ___security_cookie
00401008 XOR EAX, ESP
0040100A MOV [ESP+10H+VAR_4], EAX
0040100E XOR EAX, EAX
00401010 PUSH ESI
00401011 MOV ESI, ds:__imp__printf
00401017 PUSH OFFSET Format ; "Vpisite geslo (najvec 8 znakov): "
0040101C MOV DWORD PTR [ESP+18H+buffer], EAX
00401020 MOV DWORD PTR [ESP+18H+buffer+4], EAX
00401024 MOV [ESP+18H+buffer+8], AL
00401028 CALL ESI ; __imp__printf
0040102A ADD ESP, 4
0040102D PUSH ECX
0040102E CALL LOC_401034
00401033 DB 0E8H
00401034 LOC_401034:
00401034 POP ECX
00401035 ADD ECX, 8
00401038 PUSH ECX
00401039 RETN
0040103A DW 59E8H
0040103C DB 0FFH, 15H
0040103E DD OFFSET __imp____iob_func
00401042 DW 8D50H
00401044 DD 6A082444H, 15FF5009H
0040104C DD OFFSET __imp__fgets
00401050 DWORD_401050 DD 660CC483H, 0E8609CH, 5A000000H, 6604EBH,
00401050 DD 4A8B0000H
00401050 DD 34C03103H, 74C98555H, 0A443007H, 0F5EB491CH, 9D666141H
00401050 DD 5554BD06H, 0EBD5555H, 65D96D6H, 0DF0EBD96H, 6F517119H
00401050 DD 20537119H, 7101DF61H, 71016F50H, 0DF7F2152H, 6F5D7111H
00401050 DD 205C7111H, 7119DF75H, 71196F5FH, 543215EH, 555554BDH
00401050 DD 0D60DBD55H, 96055D95H
004010BC MOV EBP, 65693D0DH
004010C1 ADC EAX, 741BE55H
004010C6 MOV EBP, 55555554H
004010CB MOV EBP, 5D97D60FH
```

```
004010D0 POP ES
004010D1 XCHG EAX, ESI
004010D2 MOV EBP, 65053D0FH
004010D7 ADC EAX, 0D683AA55H
004010DC XCHG EAX, ECX
004010DD PUSH ECX
004010DE PUSHFW
004010E0 PUSHA
004010E1 CALL $+5
004010E6 POP EDX
004010E7 JMP SHORT LOC_4010ED
004010E9 DB 9AH, 2 DUP(0FFH)
004010EC DB 0FFH
004010ED LOC_4010ED:
004010ED MOV ECX, DS:(DWORD_401050+0EH - 40105BH)[EDX]
004010F0 XOR EAX, EAX
004010F2 XOR AL, 55H
004010F4 LOC_4010F4:
004010F4 TEST ECX, ECX
004010F6 JZ SHORT LOC_4010FF
004010F8 XOR BYTE PTR DS:(DWORD_401050+3 - 40105BH)[EDX+ECX], AL
004010FC INC ECX
004010FD JMP SHORT LOC_4010F4
004010FF LOC_4010FF:
004010FF INC ECX
00401100 POPA
00401101 POPFW
00401103 MOV ECX, DWORD PTR [ESP+18H+buffer+8]
00401107 POP ESI
00401108 XOR ECX, ESP ; cookie
0040110A XOR EAX, EAX
0040110C CALL @__security_check_cookie@4 ; __security_check_cookie(x)
00401111 ADD ESP, 10H
00401114 RETN
```

Izpis 7.14: Analiza zbirne kode pri uporabi samoreplikacijske kode (glej 7.13)

V tem primeru nam statična analiza ne pomaga nič, saj je zbirna koda kodirana s samoreplikacijsko kodo. Tako nam preostane samo še izvajanje programa v razhroščevalniku. Izpisa 7.15 in 7.16 prikazujeta dekodiranje samoreplikacijske kode. Če primerjamo oba izpisa z izpisom 7.10, vidimo, kako dekodirna rutina dekodira zbirno kodo. Izpis 7.15 prikazuje delno dekodirano

zbirno kodo (koda od naslova 0x00401099 je enaka kodi v izpisu 7.10 od naslova 0x00401053 naprej), izpis 7.16 pa prikazuje zbirno kodo odpakirano v celoti, vendar je napačno analizirana zaradi uporabe odvečne kode.

```
00401069 TEST ECX,ECX
0040106B JE SHORT SmcTest_.00401074
0040106D XOR BYTE PTR DS:[EDX+ECX+1C],AL
00401071 DEC ECX
00401072 JMP SHORT SmcTest_.00401069
00401074 INC ECX
00401075 POPAD
00401076 POPFW
00401078 PUSH ES
00401079 MOV EBP,55555554
0040107E MOV EBP,5D96D60E
00401083 PUSH ES
00401084 XCHG EAX,ESI
00401085 MOV EBP,7119DF0E
0040108A PUSH ECX
0040108B OUTS DX,DWORD PTR ES:[EDI] ; I/O command
0040108C SBB DWORD PTR DS:[ECX+53],ESI
0040108F AND BYTE PTR DS:[ECX-21],AH
00401092 ADD DWORD PTR DS:[ECX+50],ESI
00401095 OUTS DX,DWORD PTR ES:[EDI] ; I/O command
00401096 ADD DWORD PTR DS:[ECX+52],ESI
00401099 JE SHORT SmcTest_.004010C5
0040109B MOV AL,BYTE PTR SS:[ESP+8]
0040109F CMP AL,BYTE PTR SS:[ESP+9]
004010A3 JNZ SHORT SmcTest_.004010C5
004010A5 MOV CL,BYTE PTR SS:[ESP+A]
004010A9 CMP CL,BYTE PTR SS:[ESP+B]
```

Izpis 7.15: Delno dekodirana samoreplikacijska koda

```
00401069 TEST ECX,ECX
0040106B JE SHORT SmcTest_.00401074
0040106D XOR BYTE PTR DS:[EDX+ECX+1C],AL
00401071 DEC ECX
00401072 JMP SHORT SmcTest_.00401069
00401074 INC ECX
00401075 POPAD
00401076 POPFW
```

```
00401078 PUSH EBX
00401079 CALL SmcTest_.0040107F
0040107E CALL 090393DE
00401083 PUSH EBX
00401084 RETN
00401085 CALL 248C9AE5
0040108A ADD AL,3A
0040108C DEC ESP
0040108D AND AL,6
0040108F JNZ SHORT SmcTest_.004010C5
00401091 MOV DL,BYTE PTR SS:[ESP+5]
00401095 CMP DL,BYTE PTR SS:[ESP+7]
00401099 JE SHORT SmcTest_.004010C5
0040109B MOV AL,BYTE PTR SS:[ESP+8]
0040109F CMP AL,BYTE PTR SS:[ESP+9]
004010A3 JNZ SHORT SmcTest_.004010C5
004010A5 MOV CL,BYTE PTR SS:[ESP+A]
004010A9 CMP CL,BYTE PTR SS:[ESP+B]
```

Izpis 7.16: V celoti odpakirana zbirna koda

7.3.1 Primerjava

S samoreplikacijsko kodo zaščiten program smo poskušali dekodirati tudi z odpakirniki, opisanimi v podpoglavju 4.4. Rezultati so prikazani v tabeli 7.1.

Odpakirnik	Uspešno dekodiran program	Se program izvaja	Okvirni čas odpakiranja (v sekundah)	Čas, potreben za zaznavo razhroščevalnika (v sekundah)
Universal PE Unpacker	ne	/	/	/
OllyDump	ne	/	/	/
Polyunpack	ne	/	/	/
Quick Unpacker 2.2	ne	da	4	/

Tabela 7.1: Rezultati avtomatskih odpakirnikov za program, zaščiten s samoreplikacijsko kodo

Iz rezultatov vidimo, da noben odpakirnik ni uspešno dekodiral samoreplikacijske kode. Tak rezultat je pričakovan, saj program ni pakiran, ampak je kodiran s samoreplikacijsko kodo. V našem primeru smo samoreplikacijsko kodo vključili med programsko kodo, medtem ko pakirniki uporabljajo samoreplikacijsko kodo za zaščito odpakirnih rutin. Poleg samega delovanja pakirnikov je to tudi razlog, zakaj so odpakirniki neuspešni pri dekodiranju strojne kode. QuickUnpacker je upešno izvozil program na disk, vendar je ta še zmeraj kodiran. Ostali trije odpakirniki niso uspeli izvoziti programa na disk. Neuspeh je posledica delovanja odpakirnikov, saj Universal PE Unpacker in OllyDump izvažata program na podlagi izvorne vstopne točke v določeni sekciji, kar je za naš primer neuporabno, saj samoreplikacijska koda ne menja sekcij. Še največ uspeha smo imeli s odpakirnikom Polyunpack, prav zaradi njegovega načina delovanja (glej podpoglavje 4.4). Odpakirnik ni zaznal kodirnih in dekodirnih rutin (verjetno zaradi uporabe odvečne kode), smo pa delno vplivali na njegovo izvajanje in tako na delno dekodiranje in kodiranje strojne kode. Če bi vtičnik nadgradili tako, da bi upošteval vzorce samoreplikacijske kode (glej podpoglavje 7.4), potem je velika verjetnost, da bi odpakirnik uspešno dekodiral strojno kodo. Pri tem je treba še dodati, da bi moral odpakirnik zamenjati kodirne in dekodirne rutine samoreplikacijske kode z odvečno kodo. V nasprotnem primeru bi lahko prišlo do nedelovanja (sesuvanja) dekodiranega programa, saj bi se dekodirne metode izvajale nad dekodirano strojno kodo in jo s tem kodirale. Izvedba kodirane kode pa bi imela za posledico generiranje izjeme in nedelovanje programa.

Pakiranje in kodiranje sta dve različni metodi, vendar smo naredili primerjavo z odpakirniki iz razloga, ker je Polyunpack pakirnik zaradi svojega delovanja primeren za dekodiranje samoreplikacijske kode. Ob tem je treba poudariti, da bi bilo potrebno Polyunpack še nadgraditi, da bi omogočil avtomatično dekodiranje samoreplikacijske kode.

7.4 Možne izboljšave

Naša rešitev sicer otežuje statično in dinamično analizo strojne kode, vendar je ranljiva za vsaj dve vrsti napada:

- iskanje samoreplikacijske kode s pomočjo vzorcev [25],
- napad na samoreplikacijsko kodo z omejevanjem pravic izvajanja [26].

Prva metoda napada je iskanje kodirnih in dekodirnih rutin samoreplikacijske kode. Blok zakodirane programske kode nato dekodiramo z dekodirno rutino, zakodirano kodo pa zamenjamo z dekodirano. Kodirno in dekodirno rutino nadomestimo z odvečno kodo (najpreprostejše je uporaba NOP ukaza). Tako dobimo dekodiran program, ki ga lahko analiziramo in spreminjamo. Problem rešimo tako, da vpeljemo polimorfizem in metamorfizem. S tema dvema tehnikama onemogočimo iskanje vzorcev kodirnih in dekodirnih rutin, saj je vsaka kodirna in dekodirna rutina drugačna.

Druga metoda izkorišča pravice izvajanja samoreplikacijske kode [26]. Celoten program se izvaja v emulatorju, kjer se najprej sekciji, ki vsebuje strojno kodo omejijo pravice pisanja. Ko program začne izvajati samoreplikacijsko kodo, pride do izjeme kršitve dostopa (angl. access violation). Emulator izjemo prestreže in s pomočjo strukture SEH [26] ugotovi, kje v programu je do izjeme prišlo. Tako ugotovi kodirne in dekodirne rutine samoreplikacijske kode. Te rutine nato izvede emulator in tako dekodira programsko kodo. V tem primeru nam polimorfizem in metamorfizem ne pomagata, saj samoreplikacijska koda potrebuje pravice pisanja, ta napad pa deluje ravno na to lastnost. Rešitev za to bi bila vgradnja zaznavanja emulatorjev v kodirne in dekodirne rutine. Ker se koda v emulatorju počasneje izvaja kot pa koda v procesorju, je najlažji način ta, da merimo izvajanje kodirnih in dekodirnih rutin s pomočjo RDTSC ukaza. Ta postopek je enak kot postopek za zaznavanje razhroščevalnikov (glej izpis 4.6).

Poglavje 8

Zaključek

Obratni inženiring je proces, pri katerem želimo razkriti zakonitosti in delovanje programske opreme. Za uspešno izvajanje je potrebno podrobno poznati operacijski sistem, njegove posebnosti in strojni jezik. Z znanjem obratnega inženiringa lahko do potankosti razkrijemo programske zakonitosti, na podlagi pridobljenega znanja pa lahko samo strojno kodo tudi prilagodimo. Prav zaradi tega so začeli razvijati metode in orodja, ki otežijo izvajanje obratnega inženiringa.

V diplomski nalogi smo se osredotočili na tehniko samoreplikacijske kode, ki med svojim izvajanjem kodira in dekodira strojno kodo in spada med metode za onemogočanje obratnega inženiringa. Opisali smo tudi tehnike, ki se uporabljajo za zaznavanje razhroščevalnikov in sistemskih monitorjev. Tehnike za zaznavanje so odvisne od operacijskega sistema, na katerem se program izvaja. Za onemogočanje obratnega zbirnika smo uporabili odvečno kodo. Odvečna koda nam služi tudi za označevanje blokov programske kode, ki se bo kodirala in dekodirala s tehniko samoreplikacijske kode.

Kljub temu, da se uporablja samoreplikacijska koda za zaščito strojne kode, še zmeraj obstajajo tehnike, ki zaščito odpravijo. Tako lahko rečemo, da je popolna zaščita programske opreme nemogoča, lahko pa se oteži izvajanje obratnega inženiringa s kombinacijo tehnik, opisanih v diplomski nalogi. S tem ukrepom se oteži modificiranje aplikacij in odprava zaščitnih mehanizmov.

Slike

1.1 Zbirni jezik . . . 3
1.2 Prevajalnik . . . 4

2.1 Izvozna tabela . . . 15
2.2 Procesorski registri . . . 20
2.3 Kontrolni registre . . . 21
2.4 Razhroščevalni registri . . . 26
2.5 EFLAGS register . . . 29
2.6 Format x86 zbirnega ukaza . . . 30

3.1 Grafični vmesnik programa LordPE Deluxe . . . 36
3.2 Delovanje sistemskega monitorja . . . 37
3.3 Grafični vmesnik programov File Monitor in Registry Monitor . 38
3.4 Grafični vmesnik programa OllyDbg . . . 39
3.5 Grafični vmesnik programa WinDbg . . . 40
3.6 Grafični vmesnik programa Immunity Debugger . . . 41
3.7 Grafični vmesnik programa Softice . . . 42
3.8 Grafični vmesnik HyperDbg razhroščevalnika . . . 42
3.9 Delovanje HyperDbg razhroščevalnika . . . 43
3.10 Grafični vmesnik IDA obratnega zbirnika . . . 44
3.11 Diagram linearnega algoritma . . . 45
3.12 Diagram rekurzivnega algoritma . . . 46

4.1 Navidezni stroj . . . 59
4.2 Tipično delovanje PE pakirnika . . . 59
4.3 Konzolni vmesnik Upx 3.07 pakirnika . . . 63
4.4 Sekcije v testnem programu, pakiranim z Upx pakirnikom . . . 65
4.5 Grafični vmesnik ASProtect 1.64 pakirnika . . . 66
4.6 Sekcije v testnem programu, pakiranim z ASProtect 1.64 pakirnikom . . . 68

4.7 Grafični vmesnik PeLock 1.06 pakirnika 69
4.8 Sekcije testnega programa, pakiranega s pakirnikom PeLock 1.06 71
4.9 Grafični vmesnik pakirnika Enigma Protector 3.10 72
4.10 Sekcije testnega programa, pakiranega z Enigma Protector 3.10 73

5.1 Uporaba odvečne kode 75

6.1 Neodvisna samoreplikacijska koda 82

7.1 Relacija zaglavnih datotek razvojno-programskega paketa 93
7.2 Grafični vmesnik SMC generatorja 98
7.3 Dialog za nastavitev karakteristik sekcije 98

Tabele

2.1 Podatkovni imenik 11
2.2 Intel in AT&T sintaksa 17
2.3 Zgradba CR0 registra 23
2.4 Zgradba CR4 registra 25
2.5 Zgradba DR6 registra 27
2.6 Zgradba DR7 registra 28
2.7 Zgradba EFLAGS registra 29
2.8 Predpone x86 zbirnega ukaza 31
2.9 Predpone segmenta 31
2.10 Zbirni ukaz MOV glede na vrednost s in d bita 32
2.11 Vrednosti REG polja 33
2.12 Izjeme kombinacij polj R/M in MOD 33

4.1 Spremembe PE formata testnega programa, pakiranega z različnimi pakirniki 63
4.2 Rezultati avtomatskih odpakirnikov za Upx 3.07 pakirnik 65
4.3 Rezultati avtomatskih odpakirnikov nad ASProtect 1.64 pakirnikom 69
4.4 Rezultati avtomatskih odpakirnikov za PeLock 1.06 pakirnik . . 71
4.5 Rezultati avtomatskih odpakirnikov za Enigma Protector 3.10 pakirnik 74

5.1 Ukazi za operacije s skladom in registri 76

6.1 Kombinacije PUSH in POP ukaza za vse splošno dostopne registre 89

7.1 Rezultati avtomatskih odpakirnikov za program, zaščiten s samoreplikacijsko kodo 108

Literatura

[1] A. Singh, B. Singh, *Identifying malicious code through reverse engineering*, New York: Springer-Verlag, 2009, pogl. 1, 3, 4, 5

[2] Portable Executable File Format – A Reverse Engineer View. Dostopno na: `http://xylithreats.free.fr/public/CBM_1_2_2006_Goppit_PE_Format_Reverse_Engineer_View.pdf`, zadnji obisk: 23. 11. 2011

[3] P. A. Carter, PC Assembly Language. Dostopno na: `http://pdos.csail.mit.edu/6.097/readings/pcasm.pdf`, zadnji obisk: 23. 11. 2011

[4] INT 21 - Dos function codes. Dostopno na: `http://www.cs.mcgill.ca/~amaloz/FreeDOS/devel_doc/DOS%20INT%2021h.pdf`, zadnji obisk: 23. 11. 2011

[5] K. R. Irvine, *Assembly language for Intel-based computers (5th Edition)*, New Jersey: Prantice hall, 2006, pogl. 3, 4, 6, 7

[6] x86 Arhitecture. Dostopno na: `http://en.wikipedia.org/wiki/X86_architecture`, zadnji obisk: 23. 11. 2011

[7] Intel 64 and IA-32 arhitectures. Dostopno na: `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, zadnji obisk: 23. 11. 2011

[8] Rings (Computer security). Dostopno na: `http://en.wikipedia.org/wiki/Ring_%28computer_security%29`, zadnji obisk: 23. 11. 2011

[9] Directive 2009/24/EC of the European parliament and of the council on the leagal protection of computer programs, *Official Jurney of the European Union*, Strasbourg, Francija, april 2009 (dostopno

na: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2009:111:0016:0022:EN:PDF, zadnji obisk: 23. 11. 2011)

[10] Software, reverse engineering and the law. Dostopno na: http://lwn.net/Articles/134642/, zadnji obisk: 23. 11. 2011

[11] OllyDbg. Dostopno na: http://en.wikipedia.org/wiki/OllyDbg, zadnji obisk: 23. 11. 2011

[12] WinDbg. From A to Z. Dostopno na: http://windbg.info/download/doc/pdf/WinDbg_A_to_Z_bw2.pdf, zadnji obisk: 23. 11. 2011

[13] Immunity: Knowing you're secure. Dostopno na: http://www.immunitysec.com/products-immdbg.shtml, zadnji obisk: 23. 11. 2011

[14] Softice. Dostopno na: http://en.wikipedia.org/wiki/SoftICE, zadnji obisk: 23. 11. 2011

[15] A. Fattori, R. Paleari, L. Martignoni, M. Monga, Dynamic and transparent analysis of commodity production systems, *International conference on automated software engineering*, str. 417 - 426, Antwerp, Belgija, september 2010 (dostopno na: http://security.dsi.unimi.it/~joystick/pubs/ase10.pdf, zadnji obisk: 23. 11. 2011)

[16] D. Bugeja, The Anti Cracking, *Workshop in information and communication technology*, str. 54 - 60, Malta, november 2008 (dostopno na: http://www.um.edu.mt/__data/assets/pdf_file/0008/51767/wict08_submission_32.pdf, zadnji obisk: 23. 11. 2011)

[17] M. Brand, C. Valli, A. Woodward, Malware Forensic: Discovery of the Intent of Deception, *Australian digital forensics conference*, str. 41 - 49, Perth, Australia, november 2010 (dostopno na: http://ro.ecu.edu.au/cgi/viewcontent.cgi?article=1074&context=adf, zadnji obisk: 23. 11. 2011)

[18] P. Cerven, *Crackproof your software*, San Francisco: No Starch Press, 2002, pogl. 8

[19] A. Balakrishnan, C. Schulze, Code obfuscation literature survey, *CS701: Construction of Compilers, tehnično poročilo*, University of Wisconsin, Madison, Wisconsin, december 2005 (dostopno na: http://pages.cs.wisc.edu/~arinib/writeup.pdf, zadnji obisk: 23. 11. 2011)

[20] Using universal PE Unpacker plug-in in IDA Pro 4.9 to unpack compressed executables. Dostopno na: `http://www.hex-rays.com/products/ida/support/tutorials/unpack_pe/unpacking.pdf`, zadnji obisk: 23. 11. 2011

[21] A survey of reverse engineering tools for the 32-bit Microsoft Windows environment. Dostopno na: `https://www.cs.drexel.edu/~spiros/teaching/CS675/asmrceFINAL.pdf`, zadnji obisk: 23. 11. 2011

[22] Royal P., Halpin, M., Dagon, D., Edmonds, R., Lee W, PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware, *Computer security applications conference*, str. 289 - 300, Miami Beach, Florida, december 2006

[23] Quick Unpack 2.2. Dostopno na: `http://qunpack.ahteam.org/?p=436#more-436`, zadnji obisk: 23. 11. 2011

[24] M. Polychronakis, K. G. Anagnostakis, E. P. Markatos, Network level polymorphic shellcode detection using emulation, *Journal in Computer Virology*, vol 2, št. 4, str. 257 - 274, 2007

[25] N. Mavrogiannopoulos, N. Kisserli, B. Preneel, A taxonomy of self-modifying code for obfuscation, *Computer & Security*, vol. 30, str. 679 - 691, 2011

[26] Y. Wu, Z. Zhao, T. Wei Chui, An attack on SMC-based software protection, *Information and communications security conference*, str. 352 - 368, Raleigh, december 2006