Journal of Mobile, Embedded and Distributed Systems, vol. IV, no. 4, 2012
ISSN 2067 – 4074

Binary Code Disassembly for Reverse Engineering

Marius POPA
Department of Economic Informatics and Cybernetics
Bucharest University of Economic Studies
ROMANIA
marius.popa@ase.ro

Abstract: The disassembly of binary file is used to restore the software application code in a readable and understandable format for humans. Further, the assembly code file can be used in reverse engineering processes to establish the logical flows of the computer program or its vulnerabilities in real-world running environment. The paper highlights the features of the binary executable files under the x86 architecture and portable format, presents issues of disassembly process of a machine code file and intermediate code, disassembly algorithms which can be applied to a correct and complete reconstruction of the source file written in assembly language, and techniques and tools used in binary code disassembly.

Key-Words: disassembly, reverse engineering, native, intermediate code.

1 Binary code and file formats

The modern computer programs are developed in programming languages that are a human readable form [2], [3], [4], [5]. The source code written by software developers is compiled into a binary format. In software development, there are two classes of binaries:

- Machine code – is not directly understandable by software developer, but it is directly executed by the machine; it is generated by compiler depending on the hardware characteristics;
- Intermediate code – like machine code, is not directly understandable by software developer and is not directly executed by the machine; the executable code is obtained after an interpreting process performed by a specialized component called virtual machine; the most known and used virtual machines are Java Virtual Machine and Common Language Runtime (CLR) [10], [11].

The computer programs delivered in the machine code format are more difficult to be maintained because of the difficulty to understand the executable format. To implement the maintainance activities, the software developer need the source code and documentation. Another way to obtain the understandable form of the machine code is to convert it into assembly language.

The disassembly is the process which converts the machine code into equivalent format in assembly language. During this process the assembly instruction set mnemonics are translated into assembly instructions that can be easily read by software developers.

The practical and positive issues of the disassembly process and its results are [16]:

- Improvement of the portability for computer programs delivered in machine code format; unlike machine code, the intermediate code is portable due to its interpreting by a virtual machine which must be mandatorily installed on the host machine;
- The software developers determine the logical flows of the disassembled software application; the algorithms and other programming entities are extracted from the software application and used in other versions or programs;
- Security issues are identified and can be patched without access to the original source code;
- The old version of a computer program is completed with new functionalities and interfaces.

The effects of the disassembly process implementation are quantified in terms of

time and costs during the running of the computer program.
The disassembly process is one of the three main classes of techniques for reverse engineering of software [11].
Reverse engineering of software is the process for discovery the technological principles of a product or system based of analysis of its structure, function and operation [17].
The main problem of the reverse engineering is the intellectual propriety on software. As reverse engineering technique, the disassembly is used whether the machine code owners agree with it.
As negative issue, the disassembly process can be carried out by malicious software developers to discover the vulnerabilities and holes of the computer programs to hack them. Also, the discovered logical flows and algorithms can be used in other commercial computer programs without an agreement with the owners of the disassembled computer program.
The list of the available disassemblers includes tools for Windows like IDA Pro, PE Explorer, W32DASM, BORG Disassembler, HT Editor, diStorm64 and Linux like Bastard Disassembler, ciasdis, objdump, gdb, lida linux interactive disassembler, ldasm.
During the disassembly process, the most difficult issues is to separate the code from data, especially when data are inserted in code segment or code is inserted in data segment.
The assembly process removes the text-based identifiers and code comments. This issue together with the mix of data and code make more difficult the understanding of the assembly code obtained after the disassembly process.
The machine code is generated for a particular processor or family of processors. In addition, operating systems check that the machine code file has a valid executable file format. For example, the most known executable files are COM for CP/M and MS-DOS, Portable Executable (PE) for 32-bit and 64-bit version of Windows, Executable and Linkable Format (ELF) for Linux and versions of Unix, and Mach Object (Mach-O) for Mac OS X.

The executable file COM contains x86 instructions in binary format and has the following features:

- The binary code has not an organization format;
- The file execution starts from the first byte, after Program Segment Prefix;
- The COM file has a length less than 64KB;
- The content of the COM file is the image of the program in the memory.

Program Segment Prefix is a data structure used to store the state of a program and has the following features:

- It is loaded by operating system before the machine code stored in COM file;
- It contains data necessary to operating system;
- It has the length of 256 bytes.

The contents of segment registers for *x86* family of processors are depicted in figure 1.

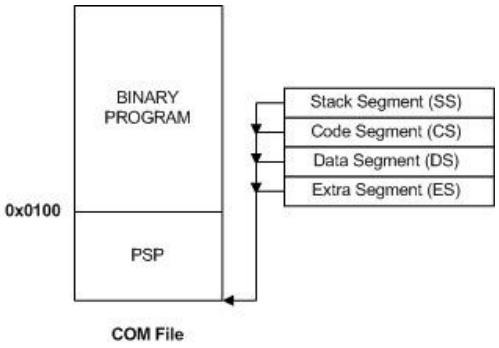

Figure 1 The contents of the segment registers for COM files

The first executed instruction has always the address CS:0x0100.
For the machine code stored in a COM file and depicted in figure 2 the disassembled code can be viewed in figure 3 when the COM file is debugged by MS-DOS application *td.exe*.

```
B80700BB090003D88BC3B8004CCD2100
0024313624
```

Figure 2 Binary executable code of the COM file

ISSN 2067 – 4074

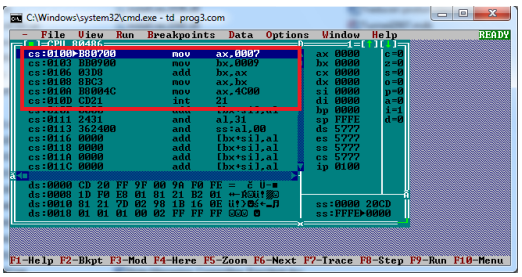

Figure 3. Disassembled code of the COM file in td.exe

After the assembly instruction **int 21h**, the next 6 bytes are used to store data in the COM file. The application *td.exe* considers the 6 bytes as operation codes for binary instructions and it tries to disassemble the bytes used for data storing. The assembly instructions generated on the 6 bytes are:

Table 1 Disassembled code from data area of the COM file

Binary code	Assembly code
0000	**add [bx+si],al**
2431	**and al,31h**
362400	**and ss:al,00h**

Because the sequence of bytes 0x3624 has not an equivalent in assembly code, td.exe application adds the next byte and the disassembled code is **and ss:al,00h.**
The executable file for Windows operating system has the following features:

- Eliminates the disadvantages of the COM files;
- Inserts a header used to identify and manage the binary code at runtime;
- Contains information regarding reallocation of the memory;
- Provides different locations for code, data and stack segments.

The contents of segment registers for x86 family of processors are depicted in figure 4.
The address of the first executed instruction is calculated using the information from the executable header.
The binary content of the x86 Windows executable file for the same logical flow like in the above COM file is depicted in figure 5.

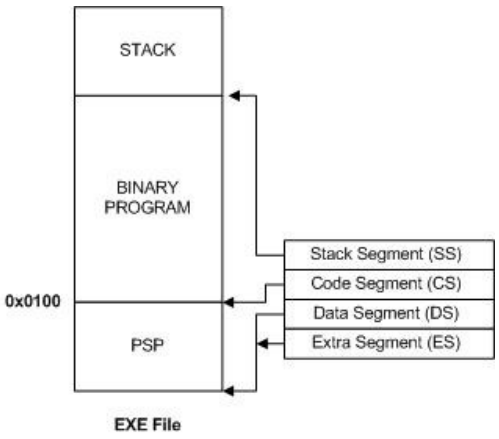

Figure 4. The content of the segment registers for Windows executable files

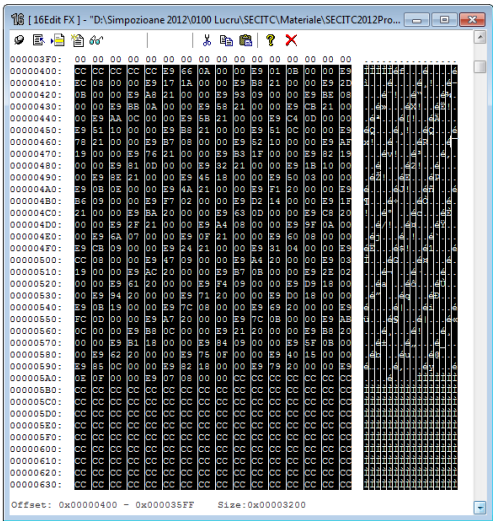

Figure 5. Binary executable code of the x86 Windows executable file

The binary executable code is included in *.text* section of the Windows exe file. The executable file has the length of 27648 bytes (27KB). The length of the *.text* section is 12799 bytes (12KB) between the address offsets 0x00000400 and 0x000035FF.
Unlike the COM file, the Windows executable file in the Portable Executable (PE) format is structured and contains metadata regarding the internal organization and code reallocation at runtime.
The PE file format structure has the following elements [1]:

1. MS-DOS information: used to keep information to MS-DOS and to treat

cross attempts to launch MS-DOS and Windows executables: it includes DOS header and MS-DOS stub program;
2. Windows information: has the role to manage the internal virtual memory space allocated for the EXE file by Windows operating system; the components are: the PE signature (the string "PE"), file header and optional header;
3. Section information: includes section headers and sections; a section has a specific type in the table 2.

Table 2 Section names in Windows PE file [13]

Name	Content
.bss	Uninitialized data (free format)
.cormeta	CLR metadata that indicates that the object file contains managed code
.data	Initialized data (free format)
.debug$F	Generated Frame Pointer Omission (FPO) debug information (object only, x86 architecture only, and now obsolete)
.debug$P	Precompiled debug types (object only)
.debug$S	Debug symbols (object only)
.debug$T	Debug types (object only)
.drective	Linker options
.edata	Export tables
.idata	Import tables
.idlsym	Includes registered Structured Exception Handler (SEH) (image only) to support Interface Definition Language (IDL) attributes.
.pdata	Exception information
.rdata	Read-only initialized data
.reloc	Image relocations
.rsrc	Resource directory
.sbss	Global Pointer (GP)-relative uninitialized data (free format)
.sdata	GP-relative initialized data (free format)
.srdata	GP-relative read-only data (free format)
.sxdata	Registered exception handler data (free format and x86/object only)
.text	Executable code (free format)
.tls	Thread-local storage (object only)
.tls$	Thread-local storage (object only)
.vsdata	GP-relative initialized data (free format and for ARM, SH4, and Thumb architectures only)
.xdata	Exception information (free format)

The section names explained in table 2 are available for binary executable files and object files under the Windows family of operating systems.
In [12], the Win32 Portable Executable file format is explained in-depth.
For 64-bit Windows system, the PE file format has few modifications aiming the widening of certain fields from 32 bits to 64 bits. The 64-bit PE file format is called PE32+.
Dynamic-Link Library (DLL) files have the same format like executable files. There is a single bit that indicates a different treatment of two kinds of file.
The content of PE file sections stored on disk is the same with the content loaded at run time into memory. PE file loading makes a mapping of PE section into the address space. Mapping makes a translation from disk offset to memory offset as it is explained in [12].
After mapping in the memory, each PE file section starts at a memory page boundary. For x86 system, the memory pages are 4 KB aligned, and 64-bit system the memory pages are 8 KB aligned.

2 Issues of Disassembly Process

Disassembly process transforms the machine code into assembly instructions readable by humans (software developer and other interested users). The main task of a disassembler tool is to identify the byte sequences corresponding to an assembly instruction.
Some features of x86 binary executables make the disassembly process more

ISSN 2067 – 4074

difficult. These features aim the following [14]:

- Code and static data can be insert in a section in a mixed manner;
- Using of variable length and unaligned instruction encodings.

The two above features are a big issue to identify the instructions hidden in or the bypass to other instruction's encoding or data bytes. So, the x86 executable format is easier to be used for hiding the malicious code in binary executables.

Identification of assembly instructions is made on code patterns delimited within the binary executable. The x86 code patterns are detailed in [16]. The structures and assembly entities are explained below.

Stack. It is a data structure used in x86 architecture to store data temporarily; the *esp* register points to the top of stack; the operating system monitors the stack to not be in a condition like underflow or overflow; the stack is a computer memory area where data are linearly stored; other memory area where data can be allocated is the heap memory; in heap, data are non-linear and variable in number and in size;

Functions and stack frames. Each function runs on its partition on the stack called stack frame; a subroutine uses the function parameters and automatic local variables allocated in the stack frame; a stack frame is created at the current *esp* location; the following assembly code is standard for a function entry:

```
push ebp
mov ebp, esp
sub esp, X
```

X represents the number of bytes allocated for the automatic variables used by the function.

The assembly code for the standard exit sequence is:

```
mov esp, ebp
pop ebp
ret
```

For the C code presented in chapter 1, the entry point in *main* function has the assembly code:

Table 3 Standard entry point of the main function

Code offset	Machine code	Assembly instructions
; void main(){		
00000	55	push ebp
00001	8B EC	mov ebp,esp
00003	81 EC E4 00 00 00	sub esp,228

The stack frame of the *main* function has 228 bytes as length.

For the same function, the standard exit sequence is:

Table 4 Standard exit sequence of the main function

Code offset	Machine code	Assembly instructions
00041	8B E5	mov esp,ebp
00043	5D	pop ebp
00044	C3	ret 0

The non-standards stack frames aim the following situations [16]:

- Using of uninitialized registers; external functions store data in registers before the subroutine calling;
- Establishing the function scope by using the *static* keyword; the external functions cannot interface with the static subroutine;
- Using other types of local variables, like static variables.

Calling conventions. They specify the rules regarding the calling of a subroutine. The rules aim the following:

- The way in which the arguments are passed to the function;
- The way in which the result or results are passed back by a function;
- The call of a function;
- Management of the stack and the stack frame by a function.

For example, for a function named *funct* having two arguments *x* and *y*, the assembly code for its call can be:

```
push x
push y
call funct
```

The *x* and *y* arguments have 32 bits, according to x86 architecture to be stored on the stack frame of the *func* function.

For example, it considers the C code for *func* function:

```
int func(int a, int b){
      int c=0;
      c=a+b;
      return c;
}
```

The assembly instructions generated from the machine code for *func* routine call written in C compiler under Visual Studio 2010 are:

Table 5 Parameter transfers and func routine call

Code offset	Machine code	Assembly instructions
00033	8B45EC	mov eax, DWORD PTR _y$[ebp]
00036	50	push eax
00037	8B4DF8	mov ecx, DWORD PTR _x$[ebp]
0003A	51	push ecx
0003B	E80000 0000	call ?func@@YAHHH@Z

Branches. In high-level programming languages, the using of *goto* instructions is recommended to be avoided. The reason is that those programming languages have been implemented the branching structures into branching instructions.
The x86 assembly language has not been implemented complex branching instructions. It uses jump instructions to control program flow.
For example, it considers the C code for the *func* routine written in C compiler under Visual Studio 2010:

```
int func(int a, int b){
      int c=0;
      if(a<b)
            c=a+b;
      else
            c=a-b;
      return c;
}
```

The disassembled code for *If-Then-Else* branch structure is:

Table 6 If-Then-Else branch structure

Code offset	Machine code	Assembly instructions
; if(a<b)		
00025	8B4508	mov eax, DWORD PTR _a$[ebp]
00028	3b450C	cmp eax, DWORD PTR _b$[ebp]
0002B	7D0B	jge SHORT $LN2@func
; c = a + b;		
0002D	8B4508	mov eax, DWORD PTR _a$[ebp]
00030	03450C	add eax, DWORD PTR _b$[ebp]
00033	8945F8	mov DWORD PTR _c$[ebp], eax
; else		
00036	eb09	jmp SHORT $LN1@func
; c = a - b;		
		$LN2@func:
00038	8B4508	mov eax, DWORD PTR _a$[ebp]
0003B	2B450C	sub eax, DWORD PTR _b$[ebp]
0003E	8945F8	mov DWORD PTR _c$[ebp], eax
; return c;		
		$LN1@func:
00041	8B45F8	mov eax, DWORD PTR _c$[ebp]

The TRUE branch is the sequence of instructions between code offsets 0x0002D and 0x00037, and the FALSE branch is delimited by the code offsets 0x00038 and 0x00040.
Avoidance of some assembly instruction blocks is possible due to using the jump instructions and labels assigned to next instruction to be executed after a jump in the logical flow of the computer program.
Loops. They are implemented for repetitive operations. To identify the loop structure in a machine code file, the following elements must be established:

- The value of condition to repeat the operation set;
- The value of condition to exit the loop structure;
- The point to start the operation set;
- The point to end the loop structure;
- The operation set.

For example, in the *func* routine written in C language under Visual Studio 2010, the *Do-For* loop is implemented:

```
int func(int a, int b){
      int c=0, i;
      for(i=1; i<=10; i++)
            c=a+b;
      return c;
}
```

After disassembling, the assembler instructions corresponding to *Do-For* loop structure are:

ISSN 2067 – 4074

Table 7 Do-For loop structure

Code offset	Machine code	Assembly instructions
; for(i=1; i<=10; i++)		
00025	C745EC 010000 00	mov DWORD PTR _i$[ebp], 1
0002C	EB09	jmp SHORT $LN3@func
		$LN2@func:
0002E	8B45EC	mov eax, DWORD PTR _i$[ebp]
00031	83C001	add eax, 1
00034	8945EC	mov DWORD PTR _i$[ebp], eax
		$LN3@func:
00037	837DEC 0A	cmp DWORD PTR _i$[ebp], 10
0003B	7F 0B	jg SHORT $LN1@func
; c=a+b;		
0003D	8B4508	mov eax, DWORD PTR _a$[ebp]
00040	03450C	add eax, DWORD PTR _b$[ebp]
00043	8945F8	mov DWORD PTR _c$[ebp], eax
00046	EBE6	jmp SHORT $LN2@func
; return c;		
		$LN1@func:
00048	8B45F8	mov eax, DWORD PTR _c$[ebp]

Besides the code patterns, the data patterns can be delimited in a binary executable. Below, some techniques to identify data in a machine code file are explained [16].

Variables. They are memory areas of a computer program where data to be processed are stored. There are classified two types of variables:

- Local variables – are defined in subroutines and are stored in stack frames; they are accessed as an offset from *esp* or *ebp*; the *static* variables are not allocated on the stack frame;
- Global variables – are accessed via a hardcoded memory address; they are not allocated in the stack and are not a limited scope.

After disassembling a machine code file, it observes that the local variables are allocated in the stack frame of a function within *.text* section, and the global variables are defined and allocated in *.data* section. The roles of *.text* and *.data* sections are explained in table 2.

The disassembled machine code for local and global variables is:

Table 8 Disassembled code for local and global variables

Code offset	Machine code	Assembly instructions
; global variable definition and allocation		
; int x = 7;		
; int y = 9;		
		PUBLIC ?x@@3HA
		PUBLIC ?y@@3HA
		_DATA SEGMENT
		?x@@3HA DD 07H
		?y@@3HA DD 09H
		_DATA ENDS
; local variable allocation		
		; int c = 0;
0001e	c745f8 000000 00	mov DWORD PTR _c$[ebp], 0
		; int i = 10;
00025	c745ec 0a0000 00	mov DWORD PTR _i$[ebp], 10

Constants. They are memory areas that do not change their content during the machine code running.

"Volatile" memory. "Volatile" variables can be accessed from external or concurrent processes. The hint to identify a "volatile" variable is a frequent access of the memory and update of its values.

Simple accessor methods. They are used to restrict the access to a variable. They receive no parameter and return the value of a variable.

Simple setter (manipulator) methods. Similar to simple accessor method, a simple setter method alters the value of a given variable.

The most part of the computer programs use complex data objects. The data structures that must be identified by a disassembler are arrays, structures and advanced structures [16].

Arrays are designed to allocate and access multiple data objects of the same type.

Structures are implemented to allocate and access data objects of different data types.

Advanced structures are implemented as support for complex operations of the computer program logical flow.

Other issues regarding the data patterns aim object-oriented programming (identification of classes and objects) and floating point numbers (using of floating point stack).

Code optimization is a stage during the compilation process. The stages of optimization are:

- Intermediate representation optimization – data flow and code flow optimizations;
- Code generation optimization – using the fast machine instructions,

During disassembly process, the control flow graph is built on sequences of instructions encoded in machine code. In [9], the control flow reconstruction is split in two parts:

- Call graph – relationship between routines are highlighted; the routines are the nodes, and the calls and returns are the edges;
- Control flow graph – jumps in the routine are highlighted, and it can be built for each routine; the nodes are called *basic blocks*, and the edges are jumps and fall-through edges; the basic blocks contain one-step executed instructions.

The reconstruction of control flow graph faces to the following problems [9]:

- Determination of the branch targets;
- Difficulties to establish the basic blocks boundaries;
- The end of a routine is difficult to be established;
- Complicated analysis because of guarded code;
- More operations assigned to instructions;
- Handling multiple entry points and external routines;
- Interlocked or overlapping procedures (optimizing compilers, hand-written assembly);
- Code blocks can contain data blocks.

The control flow graph is approximated after a static analysis on the initial control flow graph.

Compiler and link-time optimizations introduce variable instruction sequences in the machine code. This issue leads to a difficult detection of the function entry points based on patter-matching.

3. Techniques and Tools Used in Reverse Engineering

There are different techniques and tools in reverse engineering applying for the software based on Windows platforms. In [8], some of these methods are presented as it follows:

- Debugging;
- Disassembly;
- Hex-editing;
- Unpacking;
- File analysis;
- Registry monitoring;
- File monitoring.

The software developers use debuggers to fix bugs of the software under development. Debuggers are used to verify the control flows and memory area evolution during program execution for a specific test input data. These futures facilitate understanding of the algorithms and finding the content of the sensitive memory areas.

The disassembly process is presented in previous chapter together with its issues.

There are two major classes of disassembly techniques [15]:

- Static disassembly – the binary file is not executed; the instruction stream is parsed as it is found in the machine code file to establish or approximate the computer program behavior;
- Dynamic disassembly – the binary file is executed, and its execution is monitored to identify the instruction actions and behavior; the execution is made for some input sets, and as effect some instruction streams of the binary file can be avoided.

The issues of static disassembly aim the following [15]:

- Variable length instructions – as it can see in the previous chapter, the sequences of operation codes of the instructions have variable lengths; the length of each binary instruction is counted on code offsets;
- Indirect control transfers – is implemented by dynamic linking, jump tables and so forth;
- Data are interleaved with code streams – data blocks can be inserted in binary code sections making the disassembly more difficult because the disassembly tool must identify the data blocks as not being part of the binary code.

The algorithms applied in static disassembly are [15]:

ISSN 2067 – 4074

- Linear traversal disassembly – has the following features:
  - Starts at the first byte of the *.text* section; *.text* section contains the binary code of the executable as it can see in chapter 1;
  - Instructions are decoded one after another;
- Recursive traversal disassembly – consists of the following steps:
  - Starts at the first byte of the *.text* section;
  - Whenever a branch instruction is identified, the following actions are done:
    - Determination of the addresses where the branch instruction blocks begin;
    - The branch instruction blocks are disassembled;
- Other algorithms – identification of jump tables, speculative disassembly, hybrid disassembly.

The linear traversal disassembly algorithm is presented in [6], and the linear disassembly procedure has the following content:

```
while (startAddr ≤ addr ≤ endAddr) {
    I = decode instruction at address
addr;
   addr += length(I);
}
```

*) according to [6]

The linear disassembly procedure considers as input the address of the function entry point and it is executed until the end of the function calculated as:

```
endAddr = startAddr + sizeCode
```

where:

- startAddr – the address of the function entry point;
- sizeCode – length of the *.text* section;
- endAddr – the address of the function end.

The linear traversal disassembly algorithm does not take into account the control flow of the program and data embedded in the executable code.

As result, other disassembly algorithm is implemented to remove the linear disassembly disadvantages. The algorithm is presented in [6] and it has the following content:

```
while (startAddr ≤ addr ≤ endAddr){
   if (addr has been visited already)
return;
   I = decode instruction at address
addr;
   mark addr as visited;
   if (I is a branch or function call)
      for each possible target t of I do
         call disassembly rocedure for
t;
       }
   else addr += length(I);
}
```

*) according to [6]

The recursive disassembly procedure is called for the address of the function entry point, and the address of the function end calculated as with the linear disassembly procedure.

The weaknesses of the recursive traversal algorithm aim [6]:

- Assumption that the control transfer has a reasonable behavior; for example, a conditional branch has two passible targets, the function call returns to the fallowing instruction after the call instruction;
- Difficulty to identify the set of possible targets of indirect control transfers; indirect jumps are approached by ad – hoc techniques and speculative disassembly.

The disassembly algorithms works with the following elements identified or constructed on binary code [7]:

- Function entry points – functions are instruction blocks that can be independently identified and disassembled; the binary code is made by functions related to each other; the disassembly tool must identify the function entry points to bound the parts of the binary code file; identification of the function is made on instructions usually used to set up a new stack frame; also, the function call instruction can be used to identify de binary modules of the computer program;
- Control flow graph – this graph is made by nodes and edges; the nodes represent basic blocks and an edge represents a possible control flow from

a basic block to another; a basic block has not jumps or jump targets in the middle; a possible control flow is implemented by function calls, conditional or unconditional jumps, or return instructions, all these packing the control transfer instructions; a control flow graph can be built for each function; the traditional approach for intra-procedural control flow graph starts with the function entry point and instructions are disassembled until a control transfer instruction is encountered.

Because the x86 instructions have variable length and they are not aligned in memory, for each code address or code offset the disassembly algorithm tries to decode the binary code into an assembly instruction. As result, a list of potential assembly instructions is generated. A valid instruction set is extracted from the potential instruction list.

Dynamic disassembly aims snapshots of software applications at run time. Unlike static disassembly, the dynamic disassembly analyses only parts of the binary file which are to be converted into assembly code.

A static disassembler used together with debugger becomes a tool of dynamic disassembly.

In dynamic disassembly the speed of disassembly is not affected by the size of the executable file. In static disassembly, the speed of disassembly is directly proportional to the size of the executable file.

The software development technologies have evolved considering the portable requirements of the modern software applications. The code generated by such compilers has a different format from the machine code. The code is called intermediate and examples of intermediate code file are PE format for Windows-based development technologies and class type files for Java technologies.

The intermediate code is interpreted by a virtual machine at run time in order to be executed by Central Processing Unit (CPU). Also, in reverse engineering processes, the intermediate code is disassembled using software applications like Intermediate Language Disassembler (ILDASM) for Windows application or javap for Java applications.

In the below paragraphs some examples of intermediate code disassembly are offered as techniques of reverse engineering for software application that have intermediate code representation.

As NET-based disassembly example, the following C# source code is considered:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AngajatApplication
{
    class Angajat
    {
        public String Nume;
        public int id;

        public Angajat(String aNume,
int nr)
        {
            Nume = aNume;
            id = nr;
            prelDate(aNume, nr);
        }

        public String NumeAngajat()
        {
            return this.Nume;
        }

        public int IDAngajat()
        {
            return this.id;
        }

        public void prelDate(String
sNume, int snr) { }

        public static void Main() {
}
    }
}
```

The first part of the intermediate file generated by .NET compiler is presented in figure 6.

ISSN 2067 – 4074

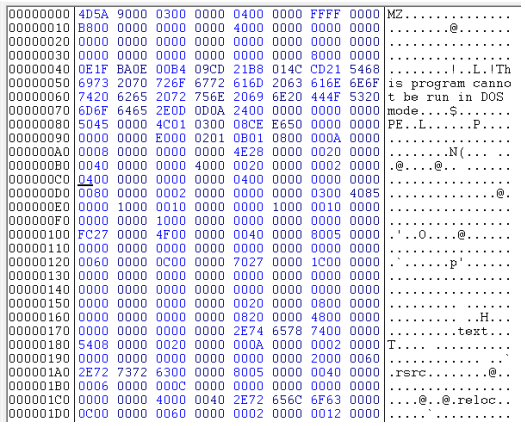

Figure 6 Intermediate code of the .NET application

In figure 6, the 0x4D5A bytes corresponding to "MZ" string in ASCII encoding and 0x5045 corresponding to "PE" string in ASCII encoding can be observed as signature of an executable file in portable format.
For NET intermediate code disassembly, the ILDASM application is used. Figure 7 highlights the .NET application loaded by ILDASM.

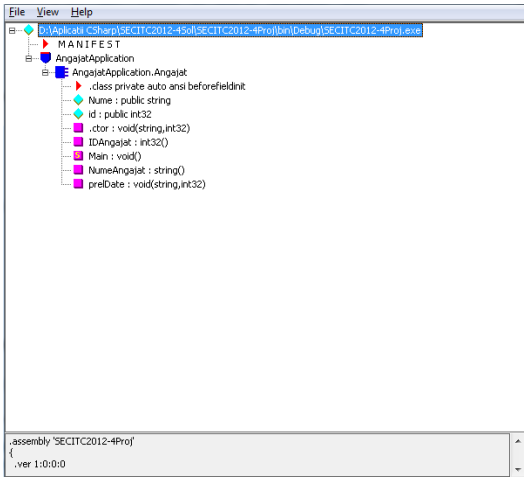

Figure 7 .NET application loaded by ILDASM disassembler

For .NET application loaded in ILDASM, the following dump options are set out:

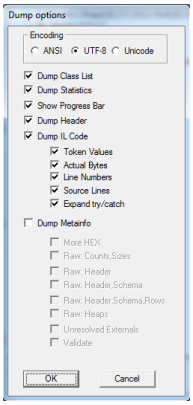

Figure 8 Dump options set out for .NET application

After dumping, a human-readable code from intermediate file is generated and the metadata assigned to PE format are presented in the restored file.
Because the restored file is very large, the below presentation contains restored code of the class Angajat.

```
// ================ CLASS MEMBERS
DECLARATION ====================

.class /*02000002*/ private auto
ansi beforefieldinit
AngajatApplication.Angajat
       extends
[mscorlib/*23000001*/]System.Object/
*01000001*/
{
  .field /*04000001*/ public string
Nume
  .field /*04000002*/ public int32
id
  .method /*06000001*/ public
hidebysig specialname rtspecialname
          instance void
.ctor(string aNume,
                                  int32
nr) cil managed
  // SIG: 20 02 01 0E 08
  {
    // Method begins at RVA 0x2050
    // Code size       33 (0x21)
    .maxstack  8
    .language '{3F5162F8-07C6-11D3-
9053-00C04FA302A1}', '{994B45C4-
E6E9-11D2-903F-00C04FA302A1}',
'{5A869D0B-6611-11D3-BD2A-
0000F80849BD}'
// Source File 'D:\Aplicatii
CSharp\SECITC2012-4Sol\SECITC2012-
4Proj\Program.cs'
    .line 13,13 : 9,45
```

```
'D:\\Aplicatii CSharp\\SECITC2012-
4Sol\\SECITC2012-4Proj\\Program.cs'
//000013:          public
Angajat(String aNume, int nr)
    IL_0000:  /* 02   |
*/ ldarg.0
    IL_0001:  /* 28   | (0A)000011
*/ call       instance void
[mscorlib/*23000001*/]System.Object/
*01000001*/::.ctor() /* 0A000011 */
    IL_0006:  /* 00   |
*/ nop
    .line 14,14 : 9,10 ''
//000014:          {
    IL_0007:  /* 00   |
*/ nop
    .line 15,15 : 13,26 ''
//000015:              Nume = aNume;
    IL_0008:  /* 02   |
*/ ldarg.0
    IL_0009:  /* 03   |
*/ ldarg.1
    IL_000a:  /* 7D   | (04)000001
*/ stfld      string
AngajatApplication.Angajat/*02000002
*/::Nume /* 04000001 */
    .line 16,16 : 13,21 ''
//000016:              id = nr;
    IL_000f:  /* 02   |
*/ ldarg.0
    IL_0010:  /* 04   |
*/ ldarg.2
    IL_0011:  /* 7D   | (04)000002
*/ stfld      int32
AngajatApplication.Angajat/*02000002
*/::id /* 04000002 */
    .line 17,17 : 13,33 ''
//000017:
prelDate(aNume, nr);
    IL_0016:  /* 02   |
*/ ldarg.0
    IL_0017:  /* 03   |
*/ ldarg.1
    IL_0018:  /* 04   |
*/ ldarg.2
    IL_0019:  /* 28   | (06)000004
*/ call       instance void
AngajatApplication.Angajat/*02000002
*/::prelDate(string,

int32) /* 06000004 */
    IL_001e:  /* 00   |
*/ nop
    .line 18,18 : 9,10 ''
//000018:          }
    IL_001f:  /* 00   |
*/ nop
    IL_0020:  /* 2A   |
*/ ret
  } // end of method Angajat::.ctor

  .method /*06000002*/ public
hidebysig instance string
          NumeAngajat() cil managed
  // SIG: 20 00 0E
  {
    // Method begins at RVA 0x2074
    // Code size       12 (0xc)
    .maxstack  1
    .locals /*11000001*/ init ([0]
string CS$1$0000)
    .line 21,21 : 9,10 ''
//000019:
//000020:          public String
NumeAngajat()
//000021:          {
    IL_0000:  /* 00   |
*/ nop
    .line 22,22 : 13,30 ''
//000022:              return
this.Nume;
    IL_0001:  /* 02   |
*/ ldarg.0
    IL_0002:  /* 7B   | (04)000001
*/ ldfld      string
AngajatApplication.Angajat/*02000002
*/::Nume /* 04000001 */
    IL_0007:  /* 0A   |
*/ stloc.0
    IL_0008:  /* 2B   | 00
*/ br.s       IL_000a

    .line 23,23 : 9,10 ''
//000023:          }
    IL_000a:  /* 06   |
*/ ldloc.0
    IL_000b:  /* 2A   |
*/ ret
  } // end of method
Angajat::NumeAngajat

  .method /*06000003*/ public
hidebysig instance int32
          IDAngajat() cil managed
  // SIG: 20 00 08
  {
    // Method begins at RVA 0x208c
    // Code size       12 (0xc)
    .maxstack  1
    .locals /*11000002*/ init ([0]
int32 CS$1$0000)
    .line 26,26 : 9,10 ''
//000024:
//000025:          public int
IDAngajat()
//000026:          {
    IL_0000:  /* 00   |
*/ nop
```

ISSN 2067 – 4074

```
    .line 27,27 : 13,28 ''
//000027:             return
this.id;
    IL_0001:  /* 02    |
*/ ldarg.0
    IL_0002:  /* 7B    | (04)000002
*/ ldfld      int32
AngajatApplication.Angajat/*02000002
*/::id /* 04000002 */
    IL_0007:  /* 0A    |
*/ stloc.0
    IL_0008:  /* 2B    | 00
*/ br.s       IL_000a

    .line 28,28 : 9,10 ''
//000028:         }
    IL_000a:  /* 06    |
*/ ldloc.0
    IL_000b:  /* 2A    |
*/ ret
  } // end of method
Angajat::IDAngajat

  .method /*06000004*/ public
hidebysig instance void
          prelDate(string sNume,
                   int32 snr) cil
managed
  // SIG: 20 02 01 0E 08
  {
    // Method begins at RVA 0x20a4
    // Code size       2 (0x2)
    .maxstack  8
    .line 30,30 : 53,54 ''
//000029:
//000030:         public void
prelDate(String sNume, int snr) { }
    IL_0000:  /* 00    |
*/ nop
    .line 30,30 : 55,56 ''
    IL_0001:  /* 2A    |
*/ ret
  } // end of method
Angajat::prelDate

  .method /*06000005*/ public
hidebysig static
          void  Main() cil managed
  // SIG: 00 00 01
  {
    .entrypoint
    // Method begins at RVA 0x20a7
    // Code size       2 (0x2)
    .maxstack  8
    .line 32,32 : 35,36 ''
//000031:
//000032:         public static void
Main() { }
    IL_0000:  /* 00    |
*/ nop
    .line 32,32 : 37,38 ''
    IL_0001:  /* 2A    |
*/ ret
  } // end of method Angajat::Main

} // end of class
AngajatApplication.Angajat
```

As Java disassembly example, the following Java source code is considered:

```
import java.*;
import java.lang.*;

class Angajat extends
java.lang.Object {

public String Nume;
public int id;

public Angajat(String aNume,int nr){
      Nume = aNume;
      id = nr;
      prelDate(aNume, nr);
}

public String NumeAngajat(){
      return this.Nume;
}

public int IDAngajat(){
      return this.id;
}

public void prelDate(String
sNume,int snr){ }
}
```

The bytecode file generated by Java compiler has the content highlighted in figure 9.

```
00000000 CAFE BABE 0003 002D 001D 0700 1007 001B 0A00 ÊÞº¾...-........
00000012 0200 0709 0001 0008 0900 0100 090A 0001 000A ................
00000024 0C00 0F00 0D0C 0017 0016 0C00 1A00 130C 001C ................
00000036 000E 0100 0328 2949 0100 1428 294C 6A61 7661 .....()I...()Ljava
00000048 2F6C 616E 672F 5374 7269 6E67 3B01 0003 2829 /lang/String;...()
0000005A 5601 0016 284C 6A61 7661 2F6C 616E 672F 5374 V...(Ljava/lang/St
0000006C 7269 6E67 3B49 2956 0100 063C 696E 6974 3E01 ring;I)V...<init>.
0000007E 0007 416E 6761 6A61 7401 000C 416E 6761 6A61 ..Angajat...Angaja
00000090 742E 6A61 7661 0100 0443 6F64 6501 0001 4901 t.java...Code...I.
000000A2 0009 4944 416E 6761 6A61 7401 000F 4C69 6E65 ..IDAngajat...Line
000000B4 4E75 6D62 6572 5461 626C 6501 0012 4C6A 6176 NumberTable...Ljav
000000C6 612F 6C61 6E67 2F53 7472 696E 673B 0100 044E a/lang/String;...N
000000D8 756D 6501 000B 4E75 6D65 416E 6761 6A61 7401 ume...NumeAngajat.
000000EA 000A 536F 7572 6365 4669 6C65 0100 0269 6401 ..SourceFile...id.
000000FC 0010 6A61 7661 2F6C 616E 672F 4F62 6A65 6374 ..java/lang/Object
0000010E 0100 0870 7265 6C44 6174 6500 2000 0100 0200 ...prelDate. .....
00000120 0000 0200 0100 1700 1600 0000 0100 1A00 1300 ................
00000132 0000 0400 0100 0F00 0E00 0100 1200 0000 3D00 ..............=.
00000144 0300 0300 0000 152A B700 032A 2BB5 0004 2A1C .......*...*+...*.
00000156 B500 052A 2B1C B600 06B1 0000 0001 0015 0000 ...*+...........
00000168 0016 0005 0000 0008 0004 0009 0009 000A 000E ................
0000017A 000B 0014 0008 0001 0014 000B 0001 0012 0000 ................
0000018C 001D 0001 0001 0000 0005 2AB4 0005 AC00 0000 ..........*.......
0000019E 0100 1500 0000 0600 0100 0000 1400 0100 1800 ................
000001B0 0C00 0100 1200 0000 1D00 0100 0100 0000 052A ...............*
000001C2 B400 04B0 0000 0001 0015 0000 0006 0001 0000 ................
000001D4 0010 0001 001C 000E 0001 0012 0000 0019 0000 ................
000001E6 0003 0000 0001 B100 0000 0100 1500 0000 0600 ................
000001F8 0100 0000 1700 0100 1900 0000 0200 11        ..............
```

Figure 9 Bytecode content of class file

After disassembly process of the class file, the restored code in the human-readable format has the following form:

```
Compiled from Angajat.java
class Angajat extends java.lang.Object {
    public java.lang.String Nume;
    public int id;
    public
Angajat(java.lang.String,int);
    public int IDAngajat();
    public java.lang.String
NumeAngajat();
    public void
prelDate(java.lang.String, int);
}

Method Angajat(java.lang.String,int)
   0 aload_0
   1 invokespecial #3 <Method
java.lang.Object()>
   4 aload_0
   5 aload_1
   6 putfield #4 <Field java.lang.String
Nume>
   9 aload_0
  10 iload_2
  11 putfield #5 <Field int id>
  14 aload_0
  15 aload_1
  16 iload_2
  17 invokevirtual #6 <Method void
prelDate(java.lang.String, int)>
  20 return

Method int IDAngajat()
   0 aload_0
   1 getfield #5 <Field int id>
   4 ireturn

Method java.lang.String NumeAngajat()
   0 aload_0
   1 getfield #4 <Field java.lang.String
Nume>
   4 areturn

Method void prelDate(java.lang.String,
int)
   0 return
```

After disassembly process, the human-readable code is analyzed to apply reverse engineering techniques or to classify the computer program as malign or benign for the computer systems.

Hex editors are software applications used to find the binary content of a file, including an executable one. A strong feature of the hex editors is permission to modify the content or to inject new content in the binary form. As effect, the behavior of the software application is observed after consecutive changes or code injections. There is hex editing software having complex functions to help its user to find quicker the executable file areas in which the user has an interest. That hex editor software can be used by any kind of user, including the users with low knowledge in software programming.

File packing is the process consisting of reduction the size of a software application, being made by a tool called file packer. At run time, software called file unpacker is launched to decompress or unpack the executable file in memory. Reverse engineering process needs the unpacked form of the executable file. A packed executable file is identifying based on its header which is modified. Manual techniques or automatic techniques like file unpacking software can be used to unpack the executable file. The main problem of the automatic techniques is to find the unpacking software to be used for a successful unpacking.

File analyzers are software used to identify the packer employed to get a packed file. Identification is made on the signature byte and it aims the compiler or programming language used to develop the packed software application.

Tools like registry monitors supervise the access to registry keys by software programs. Software application makes readings from and writings to registry keys to restore or change a configuration. Useful information for reverse engineering is obtained from the access of software application to registry keys.

File monitoring consists of supervision the access of software applications to files stored on disk. The accessed file can contain sensitive information like security algorithms used in application, access data or procedures to some functions and so forth. The file content is a valuable source of information for the reverse engineering process.

Acknowledgement

Parts of this paper were presented by the author at "5th International Conference on Security for Information Technology and Communications", Bucharest, Romania, 31 May – 1 June 2012.

ISSN 2067 – 4074

4. Conclusion

Specific techniques and tools depending on development platform and technology must be considered in order to implement a reverse engineering process. The paper content has focused on software application developed on Windows systems highlighting the specific approaching of reverse engineering for software applications developed on it.
As techniques in reverse engineering, disassembly process is used to generating the human-readable format for the computer programs delivered as machine code or intermediate code files. There are disassembly traversal algorithms to generate the assembly code from the machine code even if there is not a 100% covering of the machine code flows by the assembly code flows.
Based on the assembly language, a software specialist can implement reverse engineering techniques to investigate the software vulnerabilities of a computer program.
The main problem remains the intellectual property. Firstly, the software engineers must deal this problem with the computer program owners. On the other hand, a malicious user can break computer programs to use them for commercial advantages or to exploit their vulnerabilities to get information and other advantages unlawfully.

References

[1] Ashkbiz Danehkar, Inject your code to a Portable Executable file, 27 December 2005, http://www.codeproject.com
[2] Cătălin Boja, Security Survey of Internet Browsers Data Managers, *Journal of Mobile, Embedded and Distributed Systems – JMEDS*, vol. 3, no. 3, 2011, pp. 109 – 119
[3] Cătălin Boja, Mihai Doinea, Security Assessment of Web Based Distributed Applications, *Informatica Economică*, vol. 14, no. 1, 2010, pp. 152 – 162
[4] Cristian Toma, Security Issues for 2D Barcodes Ticketing Systems, *Journal of Mobile, Embedded and Distributed Systems – JMEDS*, vol. 3, no. 1, 2011, pp. 34 – 53
[5] Cristian Toma, Sample Development on Java Smart-Card Electronic Wallet Application, *Journal of Mobile, Embedded and Distributed Systems – JMEDS*, vol. 1, no. 2, 2009, pp. 60 – 80
[6] Cullen Linn, Saumya Debray, Obfuscation of Executable Code to Improve Resistance to Static Disassembly, *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ACM New York, NY, USA, 2003, pp. 290 – 299
[7] Giovanni Vigna, Static Disassembly and Code Analysis, *Malware Detection. Advances in Information Security*, Springer, Heidelberg, vol. 35, 2007, pp. 19 – 42
[8] Hardik Shah, Software Security and Reverse Engineering, http://www.infosecwriters.com/text_resources/pdf/software_security_and_reverse_engineering.pdf
[9] Henrik Theiling, Extracting Safe and Precise Control Flow from Binaries, *Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, IEEE Computer Society Washington, DC, USA, 2000, pp. 23 – 30
[10] Marius Popa, Techniques of Program Code Obfuscation for Secure Software, *Journal of Mobile, Embedded and Distributed Systems – JMEDS*, vol. 3, no. 4, 2011, pp. 205 – 219
[11] Marius Popa, Characteristics of Program Code Obfuscation for Reverse Engineering of Software, *Proceedings of the 4th International Conference on Security for Information Technology and Communications*, Bucharest, 17 – 18 November 2011, ASE Publishing House, Bucharest, pp. 103 – 112
[12] Matt Pietrek, An In-Depth Look into the Win32 Portable Executable File Format, msdn magazine, http://msdn.microsoft.com /en-us/magazine/cc301805.aspx
[13] Microsoft Portable Executable and Common Object File Format Specification, Revision 8.2, 21 September 2010
[14] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham,

Differentiating Code from Data in x86 Binaries, *Proceedings of the 2011 European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part III*, Springer-Verlag Berlin, Heidelberg, 2011, pp. 522 – 536

[15] Roberto Paleari, Static disassembly and analysis of malicious code, 5 July 2007, http://roberto.greyhats.it/talks.html

[16] The Wikibook of x86 Disassembly Using C and Assembly Language, Wikimedia Foundation Inc., 14 January 2008

[17] http://en.wikipedia.org/wiki/Reverse_engineering