A Survey of Reverse Engineering Tools for the 32-Bit Microsoft Windows Environment

RAYMOND J. CANZANESE, JR., MATTHEW OYER, SPIROS MANCORIDIS, and MOSHE KAM
College of Engineering
Drexel University, Philadelphia, PA, USA

Reverse engineering is defined by Chikosfky and Cross as the process of analyzing a subject system to identify the system's components and their relationships, and to create representations of the system in another form or at a higher level of abstraction. The process of reverse engineering is accomplished using specific tools that, for the 32-bit Microsoft Windows environment, are categorized as hex editors, disassemblers/debuggers, decompilers, or related technologies such as code obfuscators, unpackers, and PE editors. An evaluation of each tool is provided that identifies its domain of applicability and usability.

Categories and Subject Descriptors: A.1 [**General**]: Introductory and Survey; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Security, Documentation

Additional Key Words and Phrases: Reverse Engineering, Disassemblers, Debuggers, Decompilers, Code Obfuscators, PE Editors Unpackers, Hex Editors

1. INTRODUCTION

1.1 The Reverse Engineering Process

Software engineers are sometimes asked to understand the behavior of a program given that program's binary executable file. If they have access to the appropriate reverse engineering tools, they might choose to adhere to the following process. First, a general disassembler/debugger is used to determine the basic functionality of the program. If disassembly and debugging shows that the binary code has been obfuscated, the next step would be to determine whether the obfuscator used is a common commercial obfuscator or a custom protection scheme. A PE editor would be used to make this determination. If the obfuscator used was a common commercial obfuscator, an unpacker could be used to restore the original code. If the obfuscation has been customized, however, manual unpacking would be necessary. This is accomplished by tracing the execution of the program in a debugger until the original code is found. Then, a memory dumper can be used to write the program to the disk, and a PE editor can be used to restore the PE Headers and make the

This survey was supported in part by the Lockheed Martin Corporation in Moorestown, NJ, USA.

program executable again.

Once the obfuscation has been cracked, or if there was no obfuscation used, the process continues with a disassembler/debugger. Typically, a debugger can be used to locate code that needs to be changed, such as text strings, message box calls, or read/write functionality. To alter this code permanently, an opcode patch needs to be applied in a hex editor. Once the change has been made and applied using a hex editor, an executable will be created with the new functionality.

In some cases, using a disassembler/debugger may reveal that the code being examined was written in Delphi, Java, or Visual Basic. In the case of Java, the next step would be to use a decompiler to examine the code. In the case that the code is written in Delphi or Visual Basic, which is often made evident from the headers and the strings, a disassembler that is targeted specifically toward one of these two languages should be used. These special disassemblers can provide much more analyses of the code, and a more accurate disassembly. These programs also provide a convenient interface to alter executable files and save changes to them onto the disk.

Often, when the end of the aforementioned reverse engineering process is reached, a software engineer must return to the beginning of the process to evaluate the outcome of this process and possibly make additional changes. Sometimes, the effectiveness of these changes is compromised by anti-reverse engineering techniques. These anti-reverse engineering techniques include debugger detection, which attempts to prevent the use of reverse engineering tools, and checksums, which are computed values used to check whether changes have been made to the executable files of programs. Often, multiple iterations of this process are necessary to defeat anti-reverse engineering code, eliminate checksum checks, and so on. Figure 1 shows a flow chart of the reverse engineering process.

1.2 Categorization of Reverse Engineering Tools

Since the determination of target functionality of a binary executable is hardly practical, an arsenal of tools has been developed over the years to assist in data gathering, extraction, organization, and classification. In this survey, we concentrate on tools, which we describe in four categories, for the Microsoft Windows environment.

(1) Hex editors are programs that facilitate the editing of binary files using, typically, a hexadecimal (base-16) representation of the binary data. Hex editors are also able to display ASCII and Unicode equivalents of the hexadecimal format.

(2) Disassembler/Debuggers are programs that support the translation of hexadecimal text into assembly language, which, although human readable, is often not as intuitive as the original source code. This translation is performed by disassemblers. Debuggers can stop a program's execution at specified locations in the code to examine the program's state, thereby assisting in the comprehension of executing programs. Disassemblers and debuggers are often combined into a single tool that displays a program in assembly language and allows the execution of the program to be controlled.

(3) Decompilers attempt to translate an executable program into source code. Usu-

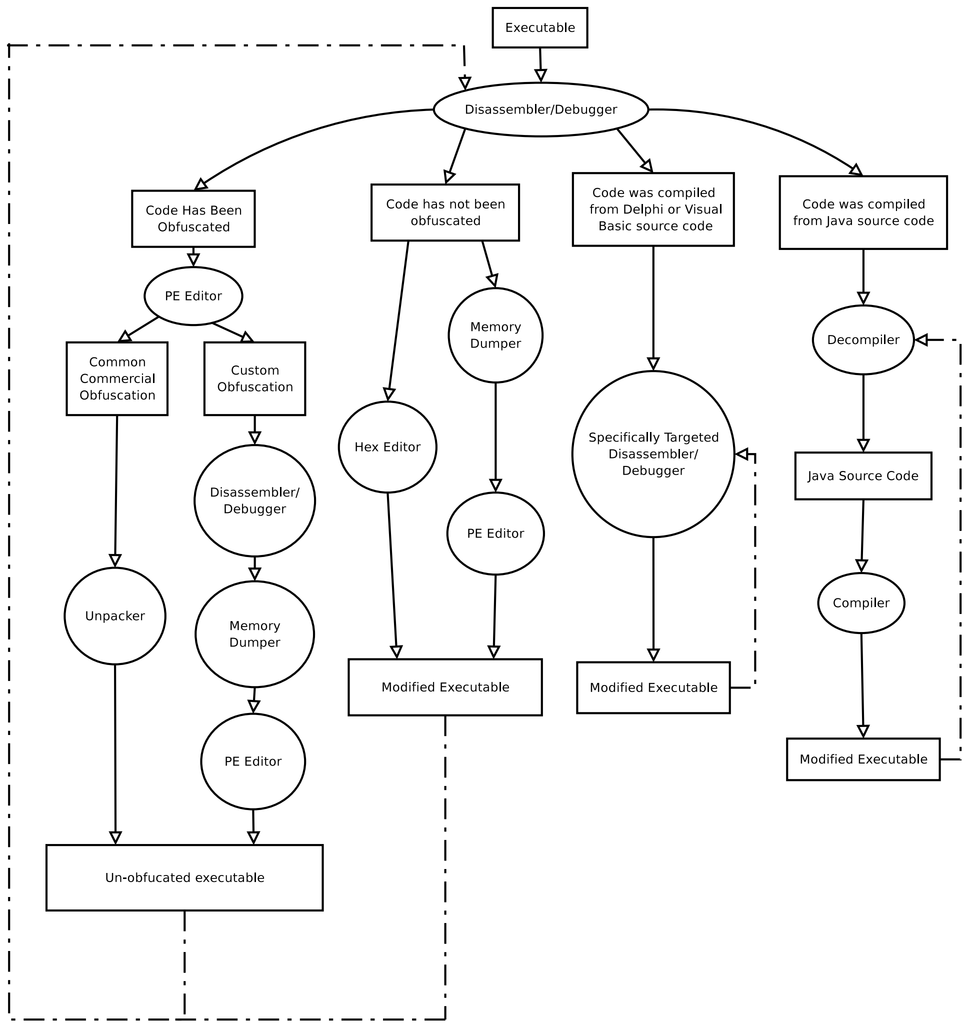


Fig. 1. The Reverse Engineering Process

ally specific to a particular compiler, decompilers also display assembly language code for parts of the program they were unable to decompile.

(4) Related technologies such as code obfuscators, PE editors, memory dumpers, and unpackers have specific applications in reverse engineering. Code obfuscators can operate on either source code or binary code in order to make the code more difficult to understand. PE editors extract the headers from PE files and allow them to be edited more efficiently than by using a hex editor. Memory dumpers allow a debugged program that has been altered in memory to be saved to disk. Unpackers are designed to overcome commercial protection techniques.

1.3 Portable Executable File Format

In order to understand the content of this paper better, one must be familiar with the Portable Executable file format. When a software engineer is provided with a target software system to dissect, he or she is often presented with a program in binary form. This paper focuses on the reverse engineering of executables designed to be run on Microsoft Windows NT/2000/XP/2003. To understand the reverse engineering process and tools better, it is important to be familiar with the basic structure of a Windows NT/2000/XP/2003 executable. Since the introduction of Windows 95, Microsoft has relied on the Portable Executable (PE) file format for executables and dynamic link libraries. The PE format is used for executables in all 32-bit versions of Windows and is the standard for Visual C++. The PE file format is designed so that an executable is loaded into memory as one contiguous block.

It is important to understand that the PE header is not the first item loaded into memory when a program is executed. A Microsoft DOS portion of the executable runs first to determine if a compatible version of Microsoft Windows is being used. The DOS header then points to the beginning of the 32-bit program, the PE Header. The PE Header contains information about how the executable was compiled, as well as the AddressOfEntryPoint (entry point), BaseOfCode (code base) and BaseOfData (data base). These items are stored as relative virtual addresses (RVAs) and point to the first line of code to be executed, the beginning of the section where the code is located, and the beginning of the section where the data used by the program is located, respectively. A relative virtual address (RVA) is the location of a line of code relative to the address of where the program begins in memory.

In addition to the PE header, each executable has a Section Table, which provides RVA's of various sections of the program, including the import table, export table, and directory table. The import table stores information about the various functions that the program calls from `DLL` files. The export table, typically used only in `DLL` files, stores information about its own functions that it allows other programs to access.

For example, message boxes, a common feature of software in the Microsoft Windows environment, are typically implemented by calling MessageBoxA, located in `KERNEL32.DLL`. The RVA of MessageBoxA would be included in the Import Table of program creating the message box and would be included in the Export Table of `KERNEL32.DLL`. `DLL` files such as this are implemented to allow multiple programs access to common functions. The Directory Table contains debug information, also stored as RVAs, including the location of the Import and Export Tables. While there are many other items included in a PE Header and Section Table, being familiar with these sections is not vital to understanding the information contained in this paper. Figure 2 shows a graphical representation of the PE file format.

We conducted a survey of the different categories of reverse engineering tools, using each tool to reverse engineer various executable programs. Our focus was to evaluate the functionality of each tool, and its applicability to the reverse engineering process.

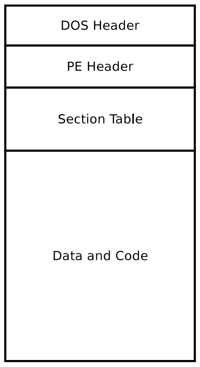


Fig. 2. PE File Format

2. HEX EDITORS

2.1 Basic Hex Editing: HHD Hex Editor

Hex editors allow users to edit binaries such as `EXE` and `DLL` files in hexadecimal form, a process known as opcode patching. Figure 3 shows this functionality. Hex editing is useful when specific instructions or strings in an executable program file need to be changed. For example, one may want to change a call for a message box to NO-OP (no operation) codes. A disassembler/debugger would be used to locate the call to the message box and the hexadecimal code equivalent of this instruction. A Microsoft Windows API reference [Microsoft Corporation 2004] is a necessary resource for recognizing these calls and their corresponding arguments. This reference includes all the functions that are native to the Microsoft Windows environment. These functions are exported from native `DLL` files such as `USER32.DLL` and `KERNEL32.DLL`.

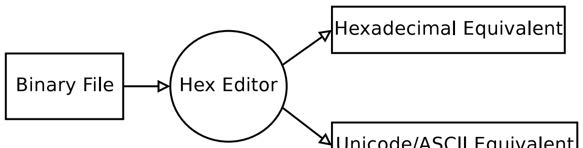


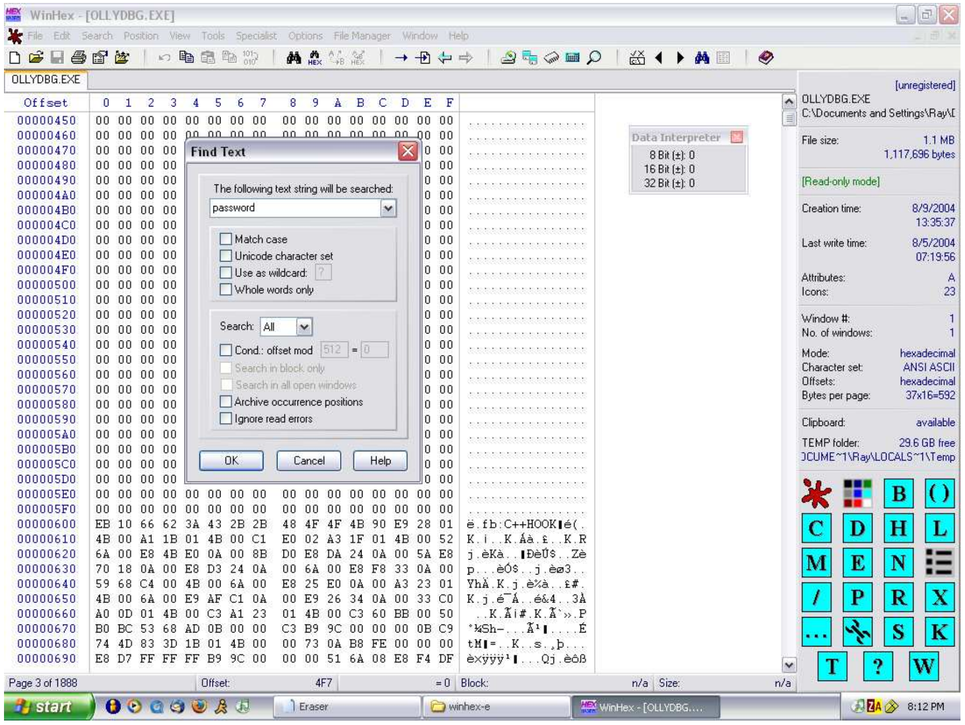
Fig. 3. The Execution of a Hex Editor

Once the corresponding hexadecimal code has been determined, the software engineer can use the search function of a hex editor to locate the code, and replace it with `90`'s, which is the hexadecimal code for NO-OP. A hex-editor can also be used to search for the strings displayed in a message box, alter them, and finally save the changes to an executable file. HHD Hex Editor [Bessonov 2000] contains all of the features that are necessary to complete these functions. It supports the viewing and editing of hexadecimal, ASCII (text) and Unicode (extended text) representations of a binary file, as well as a search feature. HHD Hex Editor also has a file comparison utility. UltraEdit [Mead 2004], another basic hex editor, features a text editor and programmer's editor in addition to basic hex editing capabilities. Several dissasemblers/debuggers, such as OllyDbg can also function as basic hex editors.

2.2 Advanced Hex Editors: WinHex, Hex Workshop, and Hackman Hex Editor

While the basic features of HHD Hex Editor or Ultra Edit are sufficient to reverse engineer most executables, sometimes a more powerful hex editor is needed. Some features common to advanced hex editors are the ability to view and edit logical and physical drives, view and edit the memory, split and merge files, perform hash calculations, and perform searches for integers and floating point values.

WinHex [Fleischmann 2004] is unique among the advanced hex editors because it features the secure deletion of files, disk cloning, and file recovery. Figure 4 shows WinHex displaying the hexadecimal and ACSII equivalents of a binary program. The search function is also displayed in the pop-up window in the center, as well as detailed information about the file and some extracted icons on the left.



```
Offset     0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000450  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000460  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000470  00 00 00 00                                 0 00  ................
00000480  00 00 00 00                                 0 00  ................
00000490  00 00 00 00                                 0 00  ................
000004A0  00 00 00 00                                 0 00  ................
000004B0  00 00 00 00                                 0 00  ................
000004C0  00 00 00 00                                 0 00  ................
000004D0  00 00 00 00                                 0 00  ................
000004E0  00 00 00 00                                 0 00  ................
000004F0  00 00 00 00                                 0 00  ................
00000500  00 00 00 00                                 0 00  ................
00000510  00 00 00 00                                 0 00  ................
00000520  00 00 00 00                                 0 00  ................
00000530  00 00 00 00                                 0 00  ................
00000540  00 00 00 00                                 0 00  ................
00000550  00 00 00 00                                 0 00  ................
00000560  00 00 00 00                                 0 00  ................
00000570  00 00 00 00                                 0 00  ................
00000580  00 00 00 00                                 0 00  ................
00000590  00 00 00 00                                 0 00  ................
000005A0  00 00 00 00                                 0 00  ................
000005B0  00 00 00 00                                 0 00  ................
000005C0  00 00 00 00                                 0 00  ................
000005D0  00 00 00 00                                 0 00  ................
000005E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000005F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000600  EB 10 66 62 3A 43 2B 2B  48 4F 4F 4B 90 E9 28 01  ë.fb:C++HOOK▮é(.
00000610  4B 00 A1 1B 01 4B 00 C1  E0 02 A3 1F 01 4B 00 52  K.¡..K.Áà.£..K.R
00000620  6A 00 E8 4B E0 0A 00 8B  D0 E8 DA 24 0A 00 5A E8  j.èKà..▮ÐèÚ$..Zè
00000630  70 18 0A 00 E8 D3 24 0A  00 6A 00 E8 F8 33 0A 00  p...èÓ$..j.èø3..
00000640  59 68 C4 00 4B 00 6A 00  E8 25 E0 0A 00 A3 23 01  YhÄ.K.j.è%à..£#.
00000650  4B 00 6A 00 E9 AF C1 0A  00 E9 26 34 0A 00 33 C0  K.j.é¯Á..é&4..3À
00000660  A0 0D 01 4B 00 C3 A1 23  01 4B 00 C3 60 BB 00 50   ..K.Ã¡#.K.Ã`».P
00000670  B0 BC 53 68 AD 0B 00 00  C3 B9 9C 00 00 00 0B C9  °¼Sh-...Ã¹▮....É
00000680  74 4D 83 3D 1B 01 4B 00  00 73 0A B8 FE 00 00 00  tM▮=..K..s.¸þ...
00000690  E8 D7 FF FF FF B9 9C 00  00 00 51 6A 08 E8 F4 DF  è×ÿÿÿ¹▮...Qj.èôß
Page 3 of 1888    Offset:    4F7    = 0    Block:    n/a    Size:    n/a
```

The following text string will be searched: password

- [ ] Match case
- [ ] Unicode character set
- [ ] Use as wildcard: ?
- [ ] Whole words only

Search: All

- [ ] Cond.: offset mod 512 = 0
- [ ] Search in block only
- [ ] Search in all open windows
- [ ] Archive occurrence positions
- [ ] Ignore read errors

8 Bit (±): 0
16 Bit (±): 0
32 Bit (±): 0

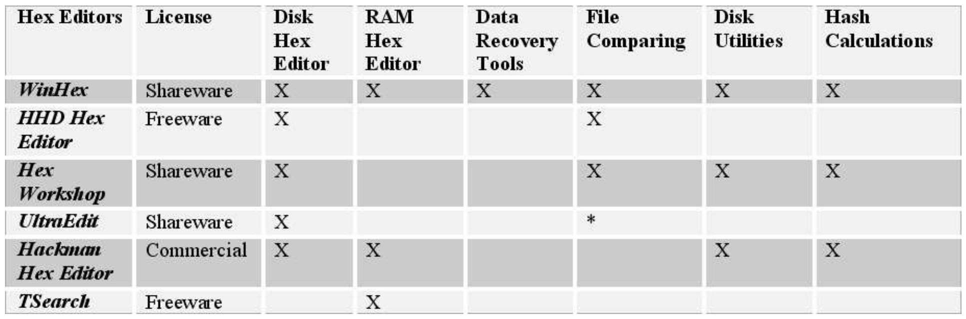
[unregistered]
OLLYDBG.EXE
C:\Documents and Settings\Ray\[

File size:	1.1 MB 1,117,696 bytes
[Read-only mode]	
Creation time:	8/9/2004 13:35:37
Last write time:	8/5/2004 07:19:56
Attributes:	A
Icons:	23
Window #:	1
No. of windows:	1
Mode:	hexadecimal
Character set:	ANSI ASCII
Offsets:	hexadecimal
Bytes per page:	37x16=592
Clipboard:	available
TEMP folder:	29.6 GB free]CUME~1\Ray\LOCALS~1\Temp

Fig. 4. Screenshot of WinHex

While WinHex is ideal for data backup, deletion, and recovery, Hex Workshop [BreakPoint Software 2004] can perform various mathematical operations and display extensive data for selected bytes. Hex Workshop's capabilities are useful when working to alter an executable that involves extensive mathematical operations either as part of its functionality or as part of its protection scheme. It can also interpret binaries written for *big endian* (i.e. PowerPC) or *little endian* (i.e. Intel's x86 architecture) applications. This refers to the byte ordering of multibyte scalar values: *Big endian* refers to the convention where the high-order byte of the number is stored in lowest memory address, while *little endian* refers to the convention where the the low-order byte of the number is stored in the lowest memory address [Peikari and Chuvakin 2004].

Table I compares various features of these popular hex editors. It compares the following features: the license of the software package (License); performing hex editing directly on the disk (Disk Hex Editing); performing hex editing operations on the memory (RAM hex editor); recovering lost files on the hard drive (Data Recovery Tools); comparing two files for differences (File Comparing); opening logical and physical drives for editing (Disk Utilities); and calculating and modifying checksums (Hash Calculations).



Hex Editors	**License**	**Disk Hex Editor**	**RAM Hex Editor**	**Data Recovery Tools**	**File Comparing**	**Disk Utilities**	**Hash Calculations**
WinHex	Shareware	X	X	X	X	X	X
HHD Hex Editor	Freeware	X			X		
Hex Workshop	Shareware	X			X	X	X
UltraEdit	Shareware	X			*		
Hackman Hex Editor	Commercial	X	X			X	X
TSearch	Freeware		X				

Table I. Comparison of Major Features of Hex Editors (* – functionality included in a seperately distributed program)

3. DISASSEMBLERS/DEBUGGERS

A disassembler's main function is to convert binary code into assembly code. Disassemblers also extract strings, used libraries, and imported and exported functions. Additionally, disassemblers typically provide heuristic analyses of the disassembled code such as locating loops, calls, and other structures. Debuggers expand the functionality of disassemblers by supporting the viewing of the stack, the CPU registers, and the hex dumping of the program as it executes. Debuggers allow breakpoints to be set and the assembly code to be edited at runtime. The applications of disassemblers/debuggers include: manually unpacking software, breaking custom protection schemes, and checking a program for faults. Using a disassembler/debugger can also provide a user with insight into the structure of the program, a feature that is especially useful when trying to modify or replicate the functionality of a program. Disassemblers/debuggers are also useful for finding passwords, creating key generators, and removing message box nag screens. Figure 5 illustrates the execution of a disassembler and a debugger.

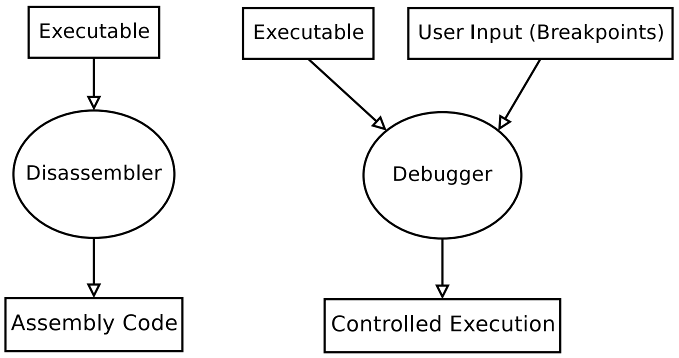


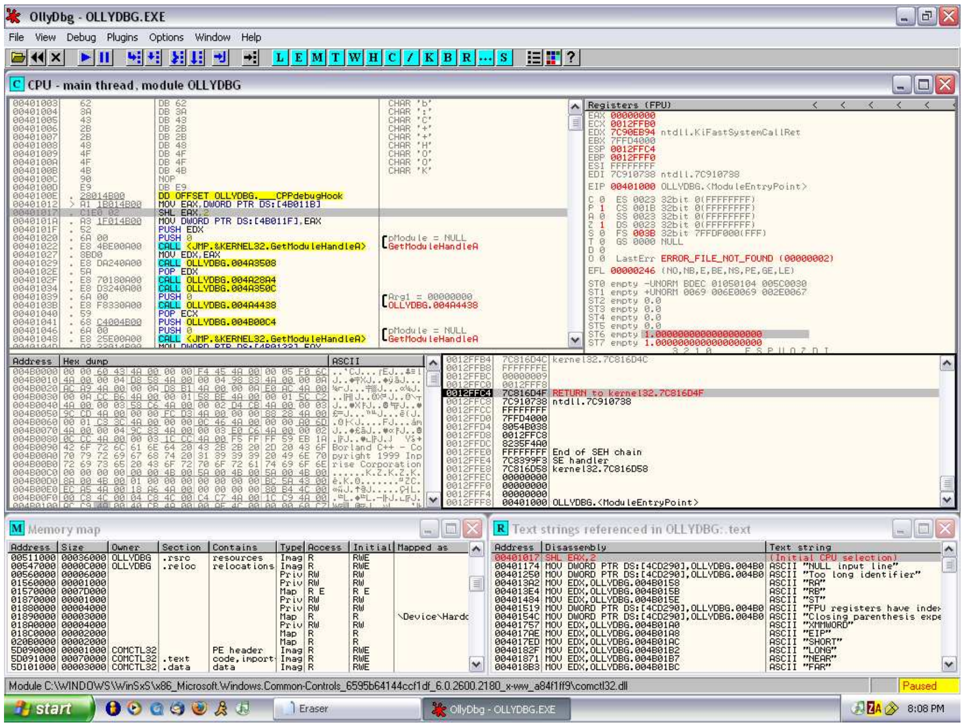
Fig. 5. Execution of a Dissasembler and Debugger

3.1 Current State of Art: OllyDbg and IDA Pro

OllyDbg [Yuschuk 2004] is an application-level debugger. Application-level debuggers are programs that are used to debug applications, and do not allow the debugging of system-level processes. OllyDbg features an interface to facilitate the simultaneous viewing of the disassembly, hex dump, stack, and CPU registers. Additionally, OllyDbg features run tracing, conditional breakpoints, PE header viewing, hex editing, and plug-in support (including memory dumping and hiding OllyDbg from the the IsDebuggerPresent API call). A shortcoming of OllyDbg is its inability to trace into SYSENTER commands called by the exception handler. Rather than trace into these calls, the debugged program continues to execute until the next exception. Figure 6 is a screenshot of OllyDbg's interface, including the disassembly with a breakpoint (top left), registers (top right), hex dump (middle left), stack (middle right), memory map (bottom left) and run trace (bottom right).

IDA Pro [DataRescue 2004] features a powerful disassembler that presents the disassembly of a binary executable program file in a color-coded and well-organized format, but lacks sufficient debugging capabilities. Like OllyDbg, IDA Pro uses arrows and brackets to the left of the disassembly to indicate jumps and loops. IDA Pro's features include hex editing, string extraction, and import and export viewing. IDA Pro also features a window for viewing all of the functions called by a program, and provides accurate analyses of the program, summarizing them in a color-coded bar at the top of the screen, which classifies the various sections of the program's code. IDA Pro's customizable kernel and processor options also facilitate accurate analyses. Although a debugger is included in one of the menus, it is a simplistic program that runs slowly and does not offer many of the features included in other debuggers, such as the ability to step through the code. Figure 7 shows IDA Pro's interface, including the disassembly and the color-coded analysis bar at the top of the screen. The titles of the other windows are visible on the tabs above the disassembly.

Two disassembler/debuggers, which we did not review, are SoftIce [Compuware Corporation 2004] and TRW [KnlSoft 2002]. These system-level debuggers are able to debug any code running in Windows, including device drivers and the kernel itself. TRW is available for Windows 9x while SoftIce runs under Windows 98/NT


OllyDbg - OLLYDBG.EXE
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
CPU - main thread, module OLLYDBG
Memory map
Text strings referenced in OLLYDBG:.text
start
Eraser
OllyDbg - OLLYDBG.EXE
8:08 PM

```
00401003    62             DB 62                                       CHAR 'b'
00401004    3A             DB 3A                                       CHAR ':'
00401005    43             DB 43                                       CHAR 'C'
00401006    2B             DB 2B                                       CHAR '+'
00401007    2B             DB 2B                                       CHAR '+'
00401008    48             DB 48                                       CHAR 'H'
00401009    4F             DB 4F                                       CHAR 'O'
0040100A    4F             DB 4F                                       CHAR 'O'
0040100B    4B             DB 4B                                       CHAR 'K'
0040100C    90             NOP
0040100D    E9             DB E9
0040100E  . 28014B00       DD OFFSET OLLYDBG.___CPPdebugHook
00401012  > A1 1B014B00    MOV EAX,DWORD PTR DS:[4B011B]
00401017  . C1E0 02        SHL EAX,2
0040101A  . A3 1F014B00    MOV DWORD PTR DS:[4B011F],EAX
0040101F  . 52             PUSH EDX
00401020  . 6A 00          PUSH 0                                      [pModule = NULL
00401022  . E8 4BE00A00    CALL <JMP.&KERNEL32.GetModuleHandleA>       [GetModuleHandleA
00401027  . 8BD0           MOV EDX,EAX
00401029  . E8 DA240A00    CALL OLLYDBG.004A3508
0040102E  . 5A             POP EDX
0040102F  . E8 70180A00    CALL OLLYDBG.004A28A4
00401034  . E8 D3240A00    CALL OLLYDBG.004A350C
00401039  . 6A 00          PUSH 0                                      [Arg1 = 00000000
0040103B  . E8 F8330A00    CALL OLLYDBG.004A4438                       [OLLYDBG.004A4438
00401040  . 59             POP ECX
00401041  . 68 C4004B00    PUSH OLLYDBG.004B00C4
00401046  . 6A 00          PUSH 0                                      [pModule = NULL
00401048  . E8 25E00A00    CALL <JMP.&KERNEL32.GetModuleHandleA>       [GetModuleHandleA
```

```
Registers (FPU)
EAX 00000000
ECX 0012FFB0
EDX 7C90EB94 ntdll.KiFastSystemCallRet
EBX 7FFD4000
ESP 0012FFC4
EBP 0012FFF0
ESI FFFFFFFF
EDI 7C910738 ntdll.7C910738

EIP 00401000 OLLYDBG.<ModuleEntryPoint>

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_FILE_NOT_FOUND (00000002)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty -UNORM BDEC 01050104 005C0030
ST1 empty +UNORM 0069 006E0069 002E0067
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 1.0000000000000000000
ST7 empty 1.0000000000000000000
```

```
Address  Hex dump                                         ASCII
004B0000 00 00 60 43 4A 00 00 00 F4 45 4A 00 00 05 F0 6C ..`CJ....EJ.....
004B0010 4A 00 00 04 D8 58 4A 00 00 04 98 83 4A 00 00 0A J....XJ.....J...
004B0020 AC A9 4A 00 00 0A D8 B1 4A 00 00 0A E0 AC 4A 00 ..J.....J.....J.
004B0030 00 0A CC B6 4A 00 00 01 58 BE 4A 00 00 01 5C C2 ....J...X.J...\.
004B0040 4A 00 00 03 58 C6 4A 00 00 02 D4 CB 4A 00 00 03 J...X.J.....J...
004B0050 9C CD 4A 00 00 00 FC D3 4A 00 00 00 88 28 4A 00 ..J.....J....(J.
004B0060 00 01 C3 3C 4A 00 00 00 0C 46 4A 00 00 00 A0 6D ...<J....FJ....m
004B0070 4A 00 00 04 9C 83 4A 00 00 03 E0 C6 4A 00 00 02 J.....J.....J...
004B0080 0C CC 4A 00 00 03 1C CC 4A 00 F5 FF FF 59 EB 1A ..J.....J....Y.+
004B0090 42 6F 72 6C 61 6E 64 20 43 2B 2B 20 2D 20 43 6F Borland C++ - Co
004B00A0 70 79 72 69 67 68 74 20 31 39 39 39 20 49 6E 70 pyright 1999 Inp
004B00B0 72 69 73 65 20 43 6F 72 70 6F 72 61 74 69 6F 6E rise Corporation
004B00C0 00 00 00 00 00 00 4B 00 5A 00 4B 00 5A 00 4B 00 ......K.Z.K.Z.K.
004B00D0 8A 00 4B 00 01 00 00 00 00 00 00 00 BC 5A 43 00 ..K..........ZC.
004B00E0 EC A5 4A 00 18 A6 4A 00 00 00 00 00 80 B4 4C 00 ..J...J.......L.
004B00F0 00 C8 4C 00 04 C8 4C 00 C4 C7 4A 00 1C C9 4A 00 ..L...L...J...J.
```

```
0012FFB4  7C816D4C kernel32.7C816D4C
0012FFB8  FFFFFFFE
0012FFBC  00000009
0012FFC0  0012FFF8
0012FFC4  7C816D4F RETURN to kernel32.7C816D4F
0012FFC8  7C910738 ntdll.7C910738
0012FFCC  FFFFFFFF
0012FFD0  7FFD4000
0012FFD4  8054B038
0012FFD8  0012FFC8
0012FFDC  8235F4A0
0012FFE0  FFFFFFFF End of SEH chain
0012FFE4  7C8399F3 SE handler
0012FFE8  7C816D58 kernel32.7C816D58
0012FFEC  00000000
0012FFF0  00000000
0012FFF4  00000000
0012FFF8  00401000 OLLYDBG.<ModuleEntryPoint>
```

```
Address  Size     Owner    Section Contains     Type Access Initial Mapped as
00511000 00036000 OLLYDBG  .rsrc   resources    Imag R      RWE
00547000 0000C000 OLLYDBG  .reloc  relocations  Imag R      RWE
00560000 00006000                               Priv RW     RW
01560000 00001000                               Priv RW     RW
01570000 0007D000                               Map  R E    R E
01870000 00001000                               Priv RW     RW
01880000 00004000                               Priv RW     RW
01890000 00003000                               Map  R      R       \Device\Hardc
018A0000 00004000                               Priv RW     RW
018C0000 00002000                               Map  R      R
020B0000 00002000                               Map  R      R
5D090000 00001000 COMCTL32         PE header    Imag R      RWE
5D091000 00070000 COMCTL32 .text   code,import  Imag R      RWE
5D101000 00003000 COMCTL32 .data   data         Imag R      RWE
```

```
Address  Disassembly                                   Text string
00401017 SHL EAX,2                                     (Initial CPU selection)
00401174 MOV DWORD PTR DS:[4CD290],OLLYDBG.004B0      ASCII "NULL input line"
00401250 MOV DWORD PTR DS:[4CD290],OLLYDBG.004B0      ASCII "Too long identifier"
004013A2 MOV EDX,OLLYDBG.004B0158                      ASCII "RA"
004013E4 MOV EDX,OLLYDBG.004B015B                      ASCII "RB"
00401484 MOV EDX,OLLYDBG.004B015E                      ASCII "ST"
00401519 MOV DWORD PTR DS:[4CD290],OLLYDBG.004B0      ASCII "FPU registers have inde
0040154C MOV DWORD PTR DS:[4CD290],OLLYDBG.004B0      ASCII "Closing parenthesis expe
00401757 MOV EDX,OLLYDBG.004B01A0                      ASCII "XMMWORD"
004017AE MOV EDX,OLLYDBG.004B01A8                      ASCII "EIP"
004017ED MOV EDX,OLLYDBG.004B01AC                      ASCII "SHORT"
0040182F MOV EDX,OLLYDBG.004B01B2                      ASCII "LONG"
00401871 MOV EDX,OLLYDBG.004B01B7                      ASCII "NEAR"
004018B3 MOV EDX,OLLYDBG.004B01BC                      ASCII "FAR"
```

```
Module C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.2180_x-ww_a84f1ff9\comctl32.dll    Paused
```

Fig. 6. Screenshot of OllyDbg

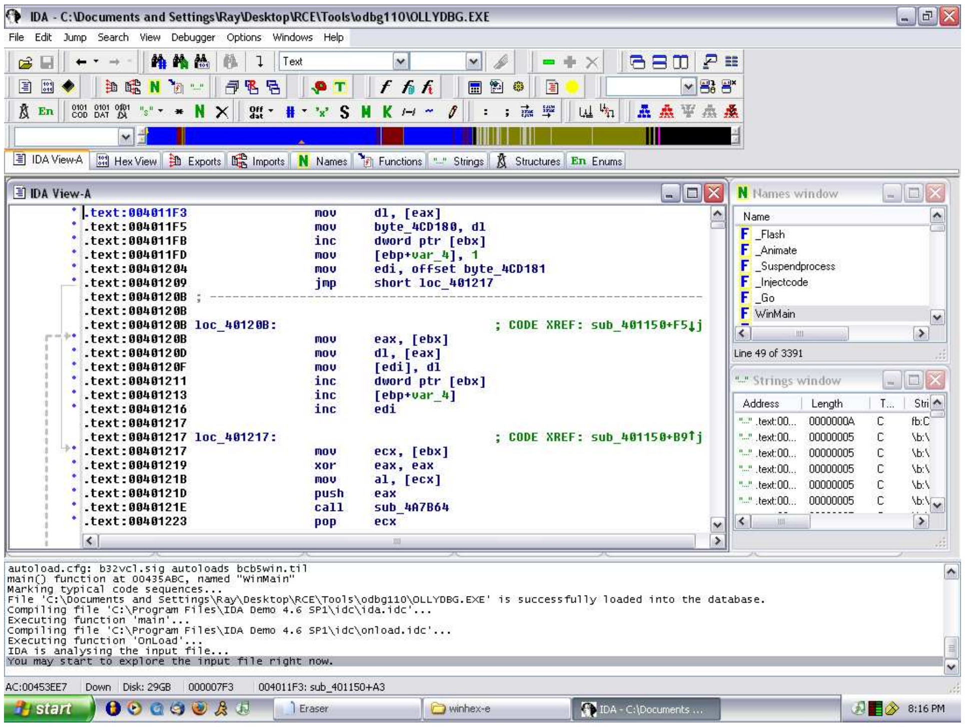
and features a remote interface. SoftIce is a commercial software library with extensive functionality but is also the target of much anti-debugging code. For this reason, lesser-known debuggers are often more practical to use.

3.2 Other Disassemblers/Debuggers

While OllyDbg and IDA Pro are two of the most fully-featured disassembler/debuggers, it is still important to have a knowledge of the disassembler/debuggers with less functionality. Like SoftIce and TRW, IDA Pro and OllyDbg are targeted by anti-debugging code, and the authors of some protection schemes specifically target the weaknesses of these debuggers.

Two debuggers feature functionality and interfaces similar to OllyDbg. The most similar in functionality is the open-source Debuggy [Fuckar 2003]. Its interface, however, is not user friendly, and its code analyses are minimal. Additionally, when running the debugger, some features of programs (even ones that are not being debugged) such as the menus, become inaccessible. W32DASM [URSoftware 1997], last updated in 1997, also has an interface similar to OllyDbg, but has less functionality than Debuggy. W32DASM lacks the color-coding and analyses that OllyDbg provides.

BORG [Cronos 2001] is a disassembler without a debugging engine. While BORG has processor options to increase the accuracy of the disassembly, the program has not been updated since the introduction of the Intel Pentium II processor. The



IDA - C:\Documents and Settings\Ray\Desktop\RCE\Tools\odbg110\OLLYDBG.EXE

```
.text:004011F3                 mov     dl, [eax]
.text:004011F5                 mov     byte_4CD180, dl
.text:004011FB                 inc     dword ptr [ebx]
.text:004011FD                 mov     [ebp+var_4], 1
.text:00401204                 mov     edi, offset byte_4CD181
.text:00401209                 jmp     short loc_401217
.text:0040120B ; ---------------------------------------------------------------
.text:0040120B
.text:0040120B loc_40120B:                             ; CODE XREF: sub_401150+F5↓j
.text:0040120B                 mov     eax, [ebx]
.text:0040120D                 mov     dl, [eax]
.text:0040120F                 mov     [edi], dl
.text:00401211                 inc     dword ptr [ebx]
.text:00401213                 inc     [ebp+var_4]
.text:00401216                 inc     edi
.text:00401217
.text:00401217 loc_401217:                             ; CODE XREF: sub_401150+B9↑j
.text:00401217                 mov     ecx, [ebx]
.text:00401219                 xor     eax, eax
.text:0040121B                 mov     al, [ecx]
.text:0040121D                 push    eax
.text:0040121E                 call    sub_4A7B64
.text:00401223                 pop     ecx
```

Address	Length	T...	Stri
.text:00...	0000000A	C	fb:C
.text:00...	00000005	C	\b:\
.text:00...	00000005	C	\b:\
.text:00...	00000005	C	\b:\
.text:00...	00000005	C	\b:\
.text:00...	00000005	C	\b:\

Line 49 of 3391

```
autoload.cfg: b32vcl.sig autoloads bcb5win.til
main() function at 00435ABC, named "WinMain"
Marking typical code sequences...
File 'C:\Documents and Settings\Ray\Desktop\RCE\Tools\odbg110\OLLYDBG.EXE' is successfully loaded into the database.
Compiling file 'C:\Program Files\IDA Demo 4.6 SP1\idc\ida.idc'...
Executing function 'main'...
Compiling file 'C:\Program Files\IDA Demo 4.6 SP1\idc\onload.idc'...
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
```

AC:00453EE7 Down Disk: 29GB 000007F3 004011F3: sub_401150+A3

Fig. 7. Screenshot of IDA Pro

disassembler still features a searchable disassembly, with separate windows to view the imports, exports, and all of the called functions. It also features a unique function to decrypt encrypted binaries.

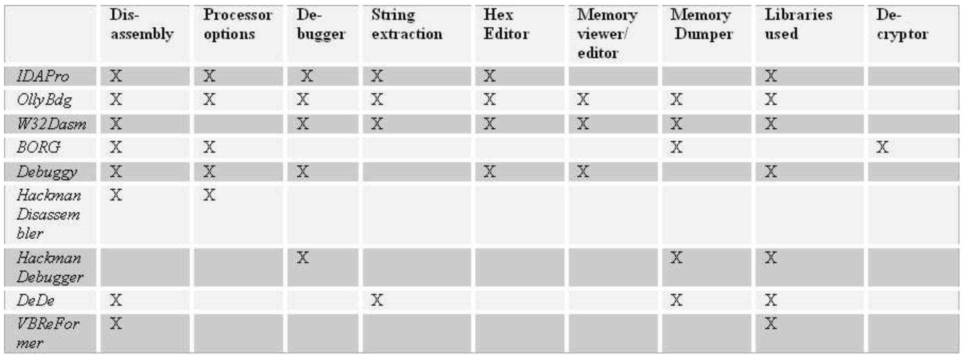
Hackman Disassembler and Hackman Debugger [TechnoLogismiki 2004] are two separately distributed tools. The disassembler performs very few analyses on the code and has an interface that is virtually identical to BORG's, but its processor options are more up to date. Hackman Debugger's debugging capabilities are limited to setting breakpoints on predetermined events, such as the loading/unloading of `DLL` files or the exit process calls. Hackman Debugger can also display the DOS Header, PE Header, and dependencies in a collapsible tree format. It also features an interface to view the imports and exports of a binary executable program file.

3.3 Programming Language-Specific Debuggers

Some disassembler/debuggers target executables compiled from a specific programming language. These disassembler/debuggers provide much more accurate analyses and make editing an executable program file easier. DeDe [DaFixer 2002] is a disassembler targeted specifically toward Delphi programs. Its tabbed interface provides information about the classes, units, forms, and procedures of a Delphi executable. DeDe has a multi-pass analysis engine, which provides for accurate analyses of an executable program, and can export its findings to project files. Additionally, DeDe can export its references to W32DASM or SoftIce. DeDe also features an OpCode-to-assembly code converter, a memory dumper, and an RVA (Relative Virtual Address) converter.

VBReFormer [Bruyere 2003] is specifically targeted toward executables compiled in Microsoft Visual Basic 5 and 6. It supports the disassembly of `EXE`, `OCX` and `DLL` files, and is able to display PE Headers. VBReformer is able to recover `VBP`, `FRM`, `CTL`, `PEG`, and `SR` files, as well as external controls, which it then displays in a windowed, color-coded environment that supports editing and saving. It also supports patching disassembled code, and is able to decompile basic code.

Table II compares the following features of some of the most popular disassemblers/debuggers: translating a binary into assembly code (Disassembly), specifying the type of processor being used (Processor Options), controling the execution of a binary (Debugger), displaying all the text strings contained in an executable (String Extraction), editing the hexadecimal equivalent of a binary (Hex Editor), viewing or editing the contents of RAM (Memory Viewer/Editor), writing a program that has been altered in memory to the disk (Memory viewer/dumper), displaying the libraries that were called by the executable (Libraries Used) and decrypting certain encrypted executables (Decryptor).



	Dis-assembly	Processor options	De-bugger	String extraction	Hex Editor	Memory viewer/ editor	Memory Dumper	Libraries used	De-cryptor
IDAPro	X	X	X	X	X			X	
OllyBdg	X	X	X	X	X	X	X	X	
W32Dasm	X		X	X	X	X	X	X	
BORG	X	X					X		X
Debuggy	X	X	X		X	X		X	
Hackman Disassem bler	X	X							
Hackman Debugger			X				X	X	
DeDe	X			X			X	X	
VBReFor mer	X							X	

Table II. Comparison of Major Features of Disassembler/Debuggers

4. DECOMPILERS

Decompilers attempt to generate high-level source code from binary code, and are targeted toward specific languages and often specific compilers. Decompilers examine the semantics of the instructions in the machine code generated by a compiler and attempt to generate source code [Cifuentes 1994b]. The decompiler generates assembly code for any code that fails to decompile. Figure 8 shows that a decompiler accepts an executable as byte code or machine code, and generates source code, which in some cases can be compiled directly back to machine code or byte code by a compiler, with only slight or no modifications. While most decompilers do not generate compileable source code, and the pseudo-source code they generate is often cryptic compared to human-written source code, the code that they supply is usually easier to understand than assembly code. The code generated by decompilers can be vital in discovering how a program works, or trying to duplicate a program's functionality.

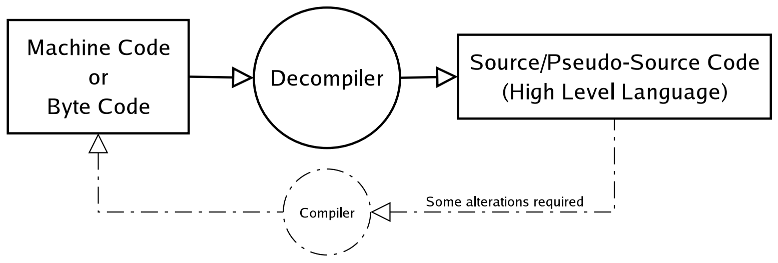


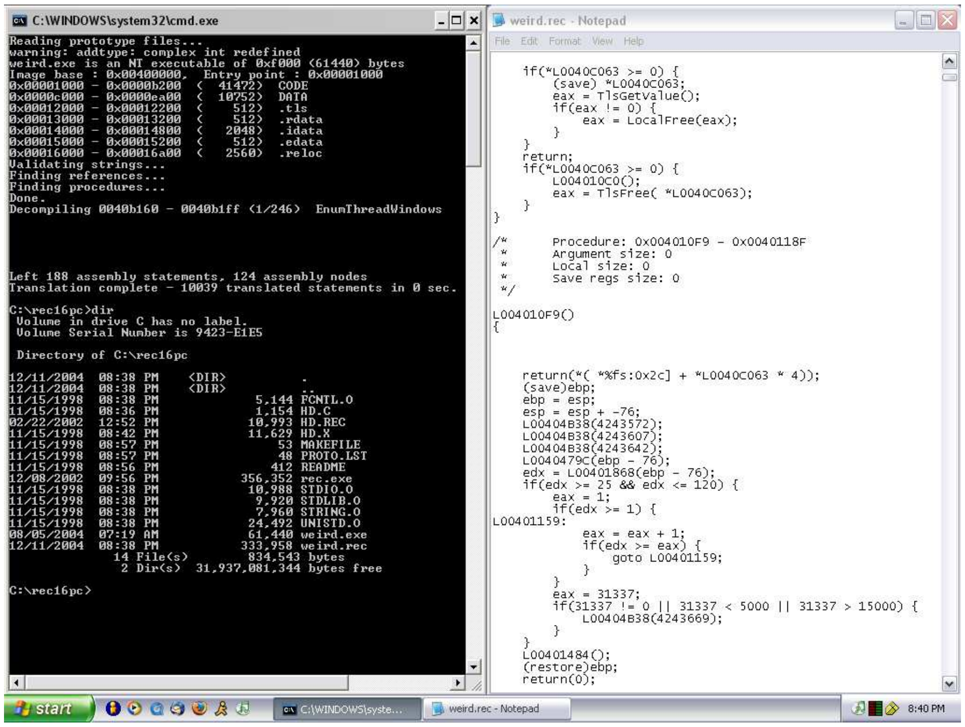
Fig. 8. Execution of a Decompiler

4.1 C Decompilers

REC [Backer Street Software 2000] is a multi-platform/multi-format decompiler. Available for Linux (i386), Windows, and SunOS, REC can decompile 386 68k, PowerPC and MIPSR3000 programs in Playstation, PE, ELF, COFF, AOUT, and CMD formats. These features make REC the most diverse decompiler we reviewed. REC also provides a command-line and HTML interface. While it works in Windows NT/2000/XP/2003, it does not disassemble the 32-bit PE format accurately unless the target was compiled with debugging information. Generally, the output code generated by REC from a 32 bit PE file without debug information is not useful.

Figure 9 shows the execution of REC in the left frame: An executable is passed to REC and is decompiled to a file with the extension REC. The right frame shows a part of this REC file. The code interacts directly with the processor's registers (ebp, ecx, etc.) and resembles assembly code rather than C source code. This is illustrative of one of the shortcomings of this, and every C decompiler we reviewed: The decompilation, even on simple files, was not complete, and assembly code was included for any code that was not disassembled. In addition, the successfully decompiled code was often poorly organized code in a cryptic format with alphanumeric identifiers. As a result, the use of a disassembler/debugger is preferred as a primary means of reverse engineering executable programs. Debuggers also have the advantage of being interactive, while decompilers simply provide static code to the user.

The two other C decompilers we evaluated, which lack the diversity of REC, are DCC and DISC. DCC [Cifuentes 1994a] is a C decompiler developed in the early 1990s and is available under the GNU Public License. It was designed to decompile i386 Microsoft DOS binaries with EXE and COM extensions into C source code. While DCC dissassembles simple Microsoft DOS binaries, it, like REC, is unable to decompile executables in the PE format accurately. DisC [Kumar 2003] is a C decompiler targeted toward binaries compiled by Borland's TurboC compiler. It is designed to decompile DOS executables to C source code, which, with only slight modifications, can be recompiled. It is able to identify library functions by



```
Reading prototype files...
warning: addtype: complex int redefined
weird.exe is an NT executable of 0xf000 (61440) bytes
Image base : 0x00400000,  Entry point : 0x00001000
0x00001000 - 0x0000b200  (  41472)  CODE
0x0000c000 - 0x0000ea00  (  10752)  DATA
0x00012000 - 0x00012200  (    512)  .tls
0x00013000 - 0x00013200  (    512)  .rdata
0x00014000 - 0x00014800  (   2048)  .idata
0x00015000 - 0x00015200  (    512)  .edata
0x00016000 - 0x00016a00  (   2560)  .reloc
Validating strings...
Finding references...
Finding procedures...
Done.
Decompiling 0040b160 - 0040b1ff (1/246)  EnumThreadWindows



Left 188 assembly statements, 124 assembly nodes
Translation complete - 10039 translated statements in 0 sec.

C:\rec16pc>dir
 Volume in drive C has no label.
 Volume Serial Number is 9423-E1E5

 Directory of C:\rec16pc

12/11/2004  08:38 PM    <DIR>          .
12/11/2004  08:38 PM    <DIR>          ..
11/15/1998  08:38 PM             5,144 FCNTL.O
11/15/1998  08:36 PM             1,154 HD.C
02/22/2002  12:52 PM            10,993 HD.REC
11/15/1998  08:42 PM            11,629 HD.X
11/15/1998  08:57 PM                53 MAKEFILE
11/15/1998  08:57 PM                48 PROTO.LST
11/15/1998  08:56 PM               412 README
12/08/2002  09:56 PM           356,352 rec.exe
11/15/1998  08:38 PM            10,988 STDIO.O
11/15/1998  08:38 PM             9,920 STDLIB.O
11/15/1998  08:38 PM             7,960 STRING.O
11/15/1998  08:38 PM            24,492 UNISTD.O
08/05/2004  07:19 AM            61,440 weird.exe
12/11/2004  08:38 PM           333,958 weird.rec
              14 File(s)        834,543 bytes
               2 Dir(s)  31,937,081,344 bytes free

C:\rec16pc>
```

```
    if(*L0040C063 >= 0) {
        (save) *L0040C063;
        eax = TlsGetValue();
        if(eax != 0) {
            eax = LocalFree(eax);
        }
    }
    return;
    if(*L0040C063 >= 0) {
        L004010C0();
        eax = TlsFree( *L0040C063);
    }
}

/*      Procedure: 0x004010F9 - 0x0040118F
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L004010F9()
{



    return(*( *%fs:0x2c] + *L0040C063 * 4));
    (save)ebp;
    ebp = esp;
    esp = esp + -76;
    L00404B38(4243572);
    L00404B38(4243607);
    L00404B38(4243642);
    L0040479C(ebp - 76);
    edx = L00401868(ebp - 76);
    if(edx >= 25 && edx <= 120) {
        eax = 1;
        if(edx >= 1) {
L00401159:
            eax = eax + 1;
            if(edx >= eax) {
                goto L00401159;
            }
        }
        eax = 31337;
        if(31337 != 0 || 31337 < 5000 || 31337 > 15000) {
            L00404B38(4243669);
        }
    }
    L00401484();
    (restore)ebp;
    return(0);
```

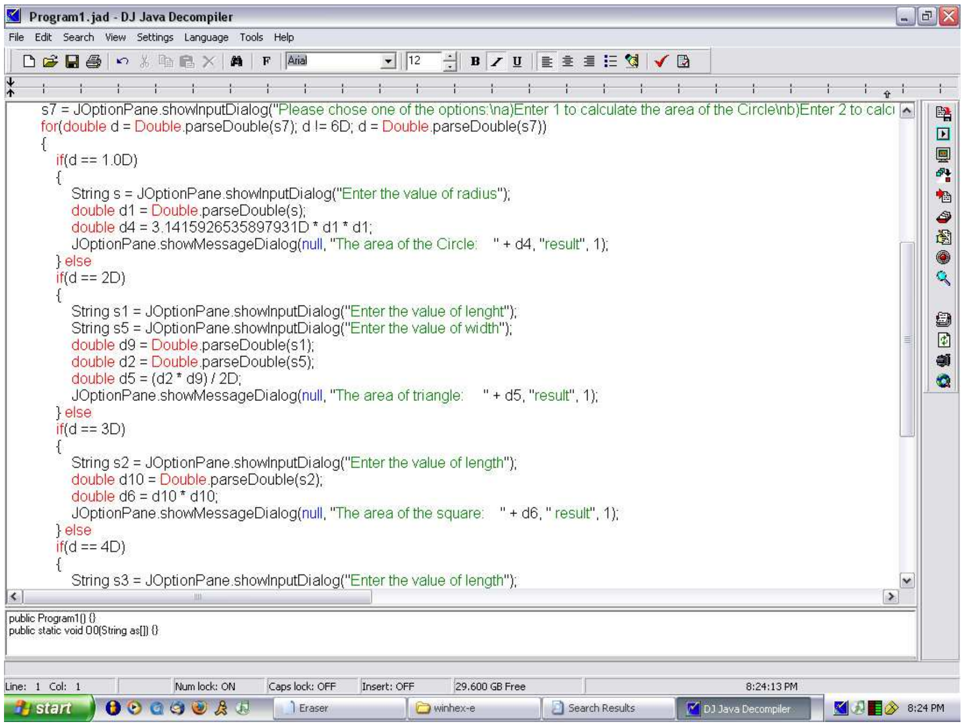
Fig. 9. Screenshot of REC(left) and Code Decompiled by REC(Right)

cross-referencing code with standard C library files and user input. The decompiler also includes a code reorganizer, which helps format the output into a readable format. DiSC also has a built-in disassembler. While it does not always decompile to compileable source, the generated code is more accurate than REC or DCC. DisC fails to recognize strings separately, and does not recognize floating point code.

4.2 Java Decompilers

Since Java files are compiled to Byte Code, which is interpreted by a Virtual Machine and not compiled to machine code like C programs, Java decompilers can decompile Java programs to compileable source code with near-perfect accuracy. As a result, Java decompilers are more widely available than C decompilers, and are also more useful. When reverse engineering an executable program written in a language that compiles into machine code (such as C), a reverse engineer relies primarily on a disassembler/debugger, due to the inaccuracy of C decompilers. When reverse engineering a Java program, however, a decompiler is a reverse engineer's primary tool. The major shortcoming of Java decompilers is that they have difficulty decompiling optimized Byte Code. The Java decompilers we surveyed were based on three core command-line java decompilers.

JAD [Kouznetsov 2001], the first of the three command-line Java decompilers, is freeware and written in C++. In our tests, JAD was able to decompile our test code into compileable source code. Known shortcomings of JAD, however, include its inability to decompile compressed ZIP or JAR archives. JAD also fails to decompile some nested loops, and has difficulty decompiling inline commands and inner functions. Because JAD is a command-line decompiler, its libraries are implemented in other front-end GUI applications, such as DJ [Neshkov 2004]. DJ features built-in compilation, applet viewing, JAR management, and exportation to HTML. The features of DJ are comparable to most other available Java decompilers based on the JAR decompiler. Figure 10 shows DJ displaying a decompiled file. Since JAD is also a code-optimizer, the code displayed is optimized, indented properly, and color-coded. JAD also supports compiling source code back to byte code.


Program1.jad - DJ Java Decompiler
File Edit Search View Settings Language Tools Help
Arial
12

```
    s7 = JOptionPane.showInputDialog("Please chose one of the options:\na)Enter 1 to calculate the area of the Circle\nb)Enter 2 to calc
    for(double d = Double.parseDouble(s7); d != 6D; d = Double.parseDouble(s7))
    {
        if(d == 1.0D)
        {
            String s = JOptionPane.showInputDialog("Enter the value of radius");
            double d1 = Double.parseDouble(s);
            double d4 = 3.1415926535897931D * d1 * d1;
            JOptionPane.showMessageDialog(null, "The area of the Circle:    " + d4, "result", 1);
        } else
        if(d == 2D)
        {
            String s1 = JOptionPane.showInputDialog("Enter the value of lenght");
            String s5 = JOptionPane.showInputDialog("Enter the value of width");
            double d9 = Double.parseDouble(s1);
            double d2 = Double.parseDouble(s5);
            double d5 = (d2 * d9) / 2D;
            JOptionPane.showMessageDialog(null, "The area of triangle:    " + d5, "result", 1);
        } else
        if(d == 3D)
        {
            String s2 = JOptionPane.showInputDialog("Enter the value of length");
            double d10 = Double.parseDouble(s2);
            double d6 = d10 * d10;
            JOptionPane.showMessageDialog(null, "The area of the square:    " + d6, " result", 1);
        } else
        if(d == 4D)
        {
            String s3 = JOptionPane.showInputDialog("Enter the value of length");
```

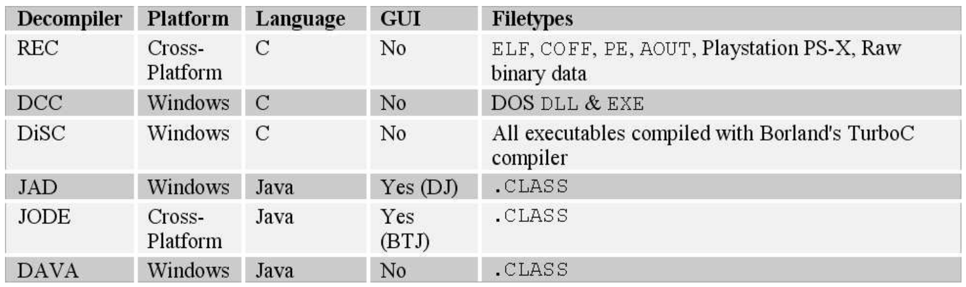
```
public Program1() {}
public static void OO(String as[]) {}
```

Fig. 10. Screenshot of DJ

JODE [Hoenicke 2004], the second command-line decompiler we reviewed, is an open-source decompiler available for use on the developer's website as an applet, and is therefore platform-independent. JODE is also a code optimizer, and the decompiled code is indented properly, but not color-coded. JODE cannot always determine all of the dependant classes. When this occurs, the decompiled code is not compileable. JODE, however, is generally more accurate than JAD. Like JAD, JODE also has various front-ends available, including Back To Java (BTJ) [CHAROLOIS 2001]. BTJ simply displays the raw classfile and Byte Code hierarchically, in addition to the decompiled source code.

DAVA [Miecznikowski 2004], the third command-line research decompiler, was developed by the Sable group at McGill University [Miecznikowski and Hendren 2001]. DAVA has been tested with other Byte Code languages, such as Haskell, Eiffel, ML, Ada, and Fortran. It performed better on these languages than JAD, while JODE's performance was comparable to DAVA's [VanEmmerik 2004]. DAVA, however, lacks a front-end such as those available for JODE or JAD. While DAVA and JODE are more accurate, JAD is the most user-friendly of the Java decompilers because of the availability of graphical front ends.

Table III compares the following features of the decompilers we evaluated: the operating systems on which they run (platform), the programming language they decompile to (language), whether they include a graphical user interface (GUI), and the filetypes they are capable of decompiling (filetypes).



Decompiler	Platform	Language	GUI	Filetypes
REC	Cross-Platform	C	No	ELF, COFF, PE, AOUT, Playstation PS-X, Raw binary data
DCC	Windows	C	No	DOS DLL & EXE
DiSC	Windows	C	No	All executables compiled with Borland's TurboC compiler
JAD	Windows	Java	Yes (DJ)	.CLASS
JODE	Cross-Platform	Java	Yes (BTJ)	.CLASS
DAVA	Windows	Java	No	.CLASS

Table III. Comparison of the Features of Decompilers

5. CODE OBFUSCATORS

Code obfuscation began as an effort to prevent the reverse engineering of software by making code less readable and more difficult to interpret. Code obfuscation techniques enable software developers to protect their intellectual property from being viewed by users or by other developers. There are two types of code obfuscation: *source code obfuscation* and *binary code obfuscation*.

5.1 Source Code Obfuscation

When a file containing source code is passed to a source code obfuscator, the obfuscator creates a file containing the obfuscated source code, as seen in Figure 11. Source code obfuscators remove indentations, comments, and spacing, and rename variables, constants, and functions. The result appears cryptic, and is difficult to read. Source code obfuscation is particularly useful when source code must be provided to a customer for compilation. Obfuscating the source code before compiling a program can also make decompilation more difficult. However, source code obfuscation typically does not complicate disassembly or debugging. Source code obfuscators are available for a variety of programming languages, such as C++ and Java.

The Semantic Designs family of Source Code Obfuscators [SemanticDesigns 2004] provide all of the basic functions of a source code obfuscator in many languages,

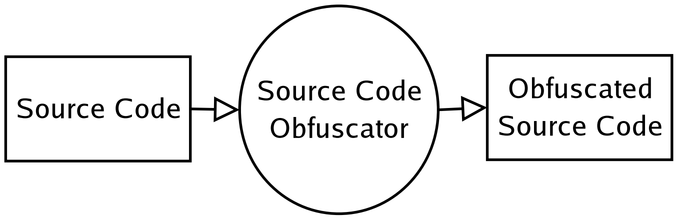
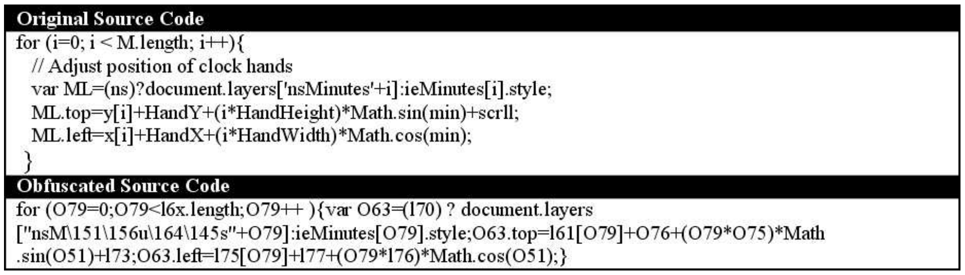


Fig. 11. Execution of a Source Code Obfuscator

including Ada, JavaScript, C, C++, Java, PHP VBScript, Verilog, and VHDL. In addition to obfuscation, these utilities help format and organize code to make it more readable to developers prior to obfuscation. This tool is also able to apply proper spacing and indentation to obfuscated code. This family of obfuscators has both a GUI and command line interface. Table IV shows an example of how obfuscated source code differs from regular source code [SemanticDesigns 2004]. This example was obfuscated with the Semantic Designs Java obfuscator. The top frame shows original source code as it was written by a programmer. The obfuscated code is displayed on the bottom. The names of the variables and functions have been replaced, and the original indentation and spacing have been removed. All this is done in an effort to make the code less readable.



Original Source Code

```
for (i=0; i < M.length; i++){
  // Adjust position of clock hands
  var ML=(ns)?document.layers['nsMinutes'+i]:ieMinutes[i].style;
  ML.top=y[i]+HandY+(i*HandHeight)*Math.sin(min)+scrll;
  ML.left=x[i]+HandX+(i*HandWidth)*Math.cos(min);
 }
```

Obfuscated Source Code

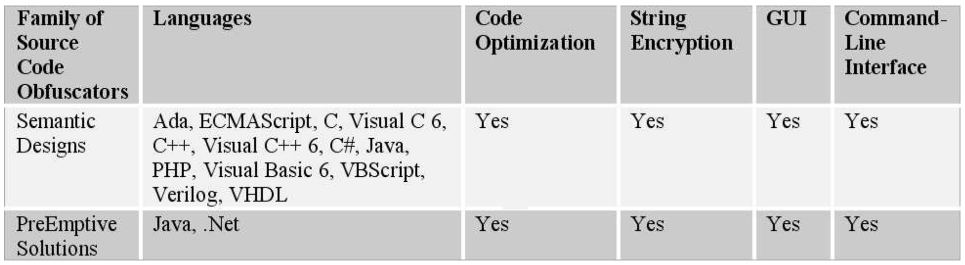
```
for (O79=0;O79<l6x.length;O79++ ){var O63=(l70) ? document.layers
["nsM\151\156u\164\145s"+O79]:ieMinutes[O79].style;O63.top=l61[O79]+O76+(O79*O75)*Math
.sin(O51)+l73;O63.left=l75[O79]+l77+(O79*l76)*Math.cos(O51);}
```

Table IV. Original Source Code vs. Obfuscated Source Code

While C source code obfuscation is a worthwhile form of software protection, source code obfuscation has a more important role to play in languages that employ a virtual machine, such as Java. As we mentioned in the previous section, decompiling programs from machine code to source code often results in cryptic pseudo-source code that is intermixed with assembly code. However, when we decompiled a Java program, the decompiler often generated compileable source code that was virtually identical to the original code, complete with indentations and proper spacing. Thus, with applications written in Virtual Machine-dependant languages such as Java, source code obfuscation appears to be more important. While obfuscation does make the decompilation more cryptic, Java decompilers such as DJ organize the code using indentation and spacing, making the code much more readable, although the changed names remain in their altered state. Figure 10 shows the decompilation of a simple Java program decompiled with DJ. The variable names are simply letters and numbers, but the strings are intact and illustrate the function of the program. This serves as an example of why both variable names and strings need to be encrypted.

PreEmptive Solutions offers an source code obfuscator targeted specifically to Java and a .NET source code obfuscator. PreEmptive Solutions's Java source code obfuscator, DashO [PreEmptive Solutions 2004], contains all of the features of a basic obfuscator, including a command line interface and GUI, with added features of code optimization and packaging. DashO also boasts an advanced renaming engine, and a string encryption feature, which is also available in Semantics' obfuscator.

Table V compares the following features of these two families of source code obfuscators: the languages that it can obfuscate (languages), whether it can optimize source code before compiling it (code optimization), whether it has the added protection of string encryption (string encryption), whether it includes a graphical user interface(GUI), and whether it contains a command-line interface (command-line).



Family of Source Code Obfuscators	Languages	Code Optimization	String Encryption	GUI	Command-Line Interface
Semantic Designs	Ada, ECMAScript, C, Visual C 6, C++, Visual C++ 6, C#, Java, PHP, Visual Basic 6, VBScript, Verilog, VHDL	Yes	Yes	Yes	Yes
PreEmptive Solutions	Java, .Net	Yes	Yes	Yes	Yes

Table V. Comparison of Features of Source Code Obfuscators

5.2 Binary Code Obfuscation: Packing and Encrypting

Binary code obfuscators accept a binary executable and a series of user-set parameters, and output a packed and/or encrypted executable, as seen in Figure 12. They also insert a decryption routine into the executable so that it can be decrypted and unpacked at run-time. Packing differs from encrypting because it also compresses the code, greatly reducing the file size of the executable program. This feature is important for programs that are to be distributed over the Internet. An obfuscated binary appears to be an undecipherable series of symbols, and only the decryption routine (or only a part of it in the case of polymorphic code) is visible on the disk. After the decryption routine has been executed, the original program becomes visible in memory. Once in memory, the program can be edited dynamically. However, in order to edit the program statically, an inline patch must be applied, or the program must be dumped from memory. The memory dumping process is complicated by anti-memory dumping code, and the fact that the PE (Portable Executable) headers and Import Address Tables will need to be restored to their original state. Figure 13 compares the structure of a packed executable as it appears statically on the disk (left) and as it appears in dynamically in memory (right). Binary code obfuscators that involve encryption and/or packing include ASProtect, Y0da's Cryptor, NFO, and Armadillo. [Cerven 2002]

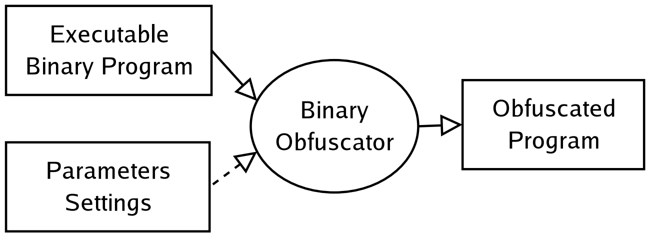


Fig. 12. Execution of a Binary Code Obfuscator

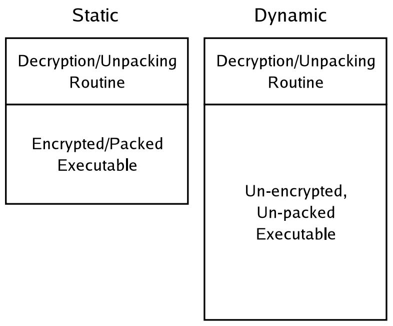
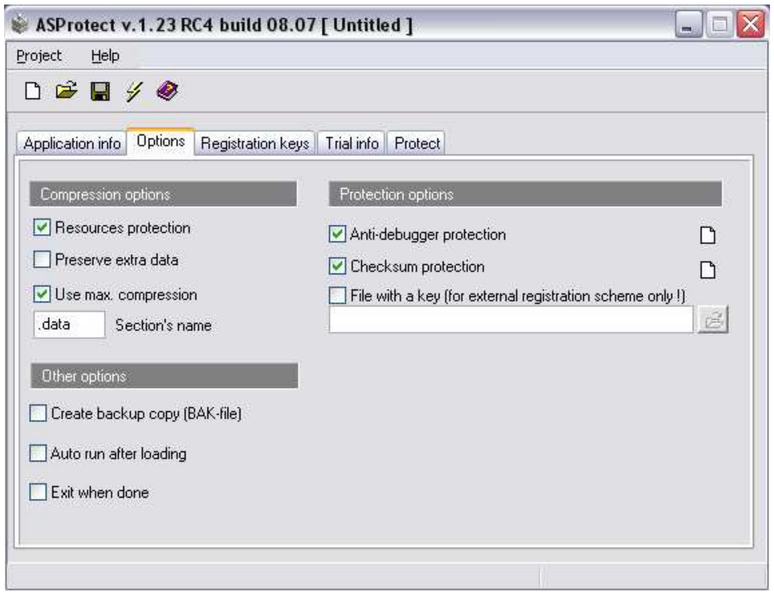

Static
Dynamic
Decryption/Unpacking Routine
Encrypted/Packed Executable
Decryption/Unpacking Routine
Un-encrypted, Un-packed Executable

Fig. 13. Static Packed Executable (left) vs. Dynamic Packed Executable(right)

ASProtect [Solodovnikov 2003] is widely used to obfuscate shareware programs or demos that are distributed through the Internet. ASProtect inserts anti-debugging code into obfuscated executables, and contains features specific to demonstrations, including time limits and registration code handling. Figure 14 shows some of the customizeable features available in the trial version of ASProtect. ASProtect also uses the Windows Structured Exception Handler to complicate run-tracing and to execute its anti-debugging code. ASProtect deters run tracing in OllyDbg, because OllyDbg is unable to trace into SYSENTER commands at the end of the exception handlers.

ASProtect also uses a technique known as "stolen bytes." Stolen bytes refers to an anti-memory dumping technique that involves the deletion of a section of code immediately after it is executed. In ASProtect, six bytes from the beginning of the program are executed before the Original Entry Point is reached, and subsequently deleted. The Original Entry Point (OEP) refers to the line of code where the program began execution prior to obfuscation. As a result of this deletion, the program will not run after being dumped unless these bytes are restored.

ASProtect can also check whether the unpacking code is in memory to determine whether the program has been dumped. ASProtect's protections, features, and customizability define the state-of-the-art of prepackaged packers/encrypters, and therefore it is one of the most difficult to reverse. Other binary code obfuscators, such as Y0da's Crypter, NFO, and Armadillo rival some of ASProtect's protections.



Compression options

[✓] Resources protection
[ ] Preserve extra data
[✓] Use max. compression
.data Section's name

Other options

[ ] Create backup copy (BAK-file)
[ ] Auto run after loading
[ ] Exit when done

Protection options

[✓] Anti-debugger protection
[✓] Checksum protection
[ ] File with a key (for external registration scheme only !)

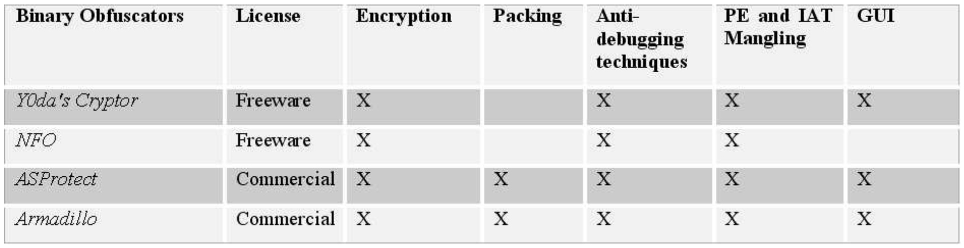
Fig. 14. Screenshot of ASProtect

Y0da's Crypter [Danehkar and Bzdok 2004] is a freeware binary obfuscator that implements CRC checksum calculations to check whether the program has been altered. While not as customizeable as ASProtect, Y0da's crypter creates executables that are difficult to reverse and is often used to protect non-commercial software.

Armadillo [Silicon Realms Toolworks 2004] provides basic protections and is not difficult to reverse. Armadillo has an optional feature that decrypts the program in parts, decrypting and executing parts of the code, then encrypting them and moving on to the next piece of code. Consequently, the entire original program is never present in memory at any point in time. This unique feature makes dumping an Armadillo-protected executable from memory difficult.

The binary obfuscator NFO [bartĈrackPl 2000] differs from ASProtect, Y0da's Crypter, and Armadillo because it does not allow any parameters to be passed to the packer. As a result, all executables that are packed with NFO are packed in an identical manner. Therefore, once the protection scheme is broken, writing an automatic unpacker is simple, and such an unpacker will work on all NFO-packed executables. NFO serves as an example of why the customizability of a protection scheme is vital to protecting it against the writing of automatic unpackers.

Table VI compares the following features of these binary obfuscators: The license of the program, whether it encrypts the executable (Encryption), whether it reduces the size of an executable (Packing), whether it protects against the use of a debugger (Anti-debugging techniques), whether it provides security against memory dumping by altering the PE headers and Import Address Table (PE and IAT Mangling) and whether it contains a GUI.



Binary Obfuscators	License	Encryption	Packing	Anti-debugging techniques	PE and IAT Mangling	GUI
Y0da's Cryptor	Freeware	X		X	X	X
NFO	Freeware	X		X	X	
ASProtect	Commercial	X	X	X	X	X
Armadillo	Commercial	X	X	X	X	X

Table VI. Comparison of Major Features of Binary Obfuscators

6. UNPACKERS

Several tools, commonly known as unpackers, have been developed to combat binary obfuscation. Unpackers are widely available on the Internet for specific protection schemes.

6.1 "Universal" Unpackers

Some programmers have attempted to create universal unpackers, which can determine the binary obfuscator used to pack a program and then unpack the program. The Generic Unpacker for Windows (GUW) [Gabler 2001], and ProcDump [G-RoM et al. 2000] are two such programs. Both fail to unpack binaries protected by complex and configurable protection schemes, such as ASProtect. Their functionality is limited to unpacking simpler protection schemes, such as Armadillo.

Programs such as PeiD [Snaker and Qwerton 2004] are able to identify specifics such as the version of the binary obfuscation package used to protect a binary. These programs, however, are designed solely to identify the protection scheme.

6.2 Packer-Specific Unpackers

Once the protection scheme is known, the decision of which unpacker to use becomes easier. Universal unpackers work well for simple protection schemes, while packer-specific unpackers or manual unpacking may be a better approach for more complex or less common protection schemes. One example of a packer-specific unpacker is unNFO [Dulek 2000], which can unpack executables obfuscated with NFO version 1.0. In all tests we conducted unNFO was successful in unpacking NFO 1.0 protected executables. This provides further evidence that binary obfuscators that offer little customizability do not provide adequate protection.

Generally, packer-specific unpackers have a greater success rate than universal unpackers, provided that the binary obfuscator used to obfuscate the executable program can be identified. Complications arise, however, when highly configurable packers are used. For example, we found that several unpackers targeted specifically toward ASProtect were unable to unpack any of the ASProtected program we had in our possession. We attributed the unpacker failures to the highly customizable nature of the protection scheme.

When automatic unpackers fail, as is often the case when ASProtect or other customizable obfuscators are used, debuggers can be used to trace a program to its original entry point, and then memory dumpers, such as those provided in OllyDbg or ProcDump, can be used to dump the program back onto the hard drive. After dumping a program from the memory however, the program's PE header and import address table will likely be corrupted and will need to be repaired in order for the dumped code to be executed. Figure 15 shows this basic framework for unpacking a file.

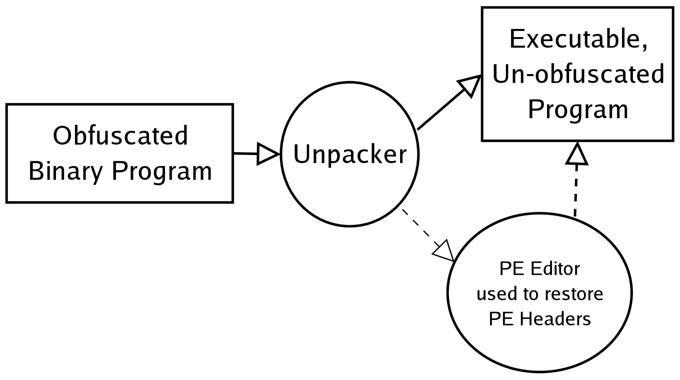

Obfuscated Binary Program
Unpacker
Executable, Un-obfuscated Program
PE Editor used to restore PE Headers

Fig. 15. Execution of an Unpacker

7. PE EDITORS

PE headers and import address tables (IAT's) need to be repaired after dumping a program from memory because the location of each item is stored as a relative virtual address (RVA). Once the program is dumped and the new entry point is set, the reference point of these addresses will be different. The addresses of all the other items will need to be changed to reflect this new reference point. To further complicate the process, some binary obfuscators encrypt the import table, alter its size, use emulated addresses, or otherwise mangle the import table. While hex editors can be used to edit the PE headers, the complexity of the PE headers makes this a laborious process. PE Editors simplify this process by allowing PE headers to be edited in an organized, graphical environment. Figure 16 illustrates the execution of a PE Editor.

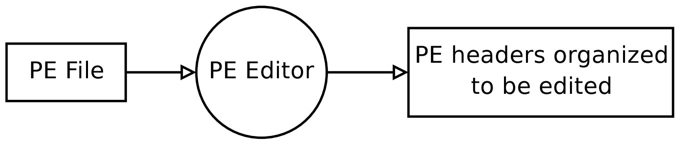
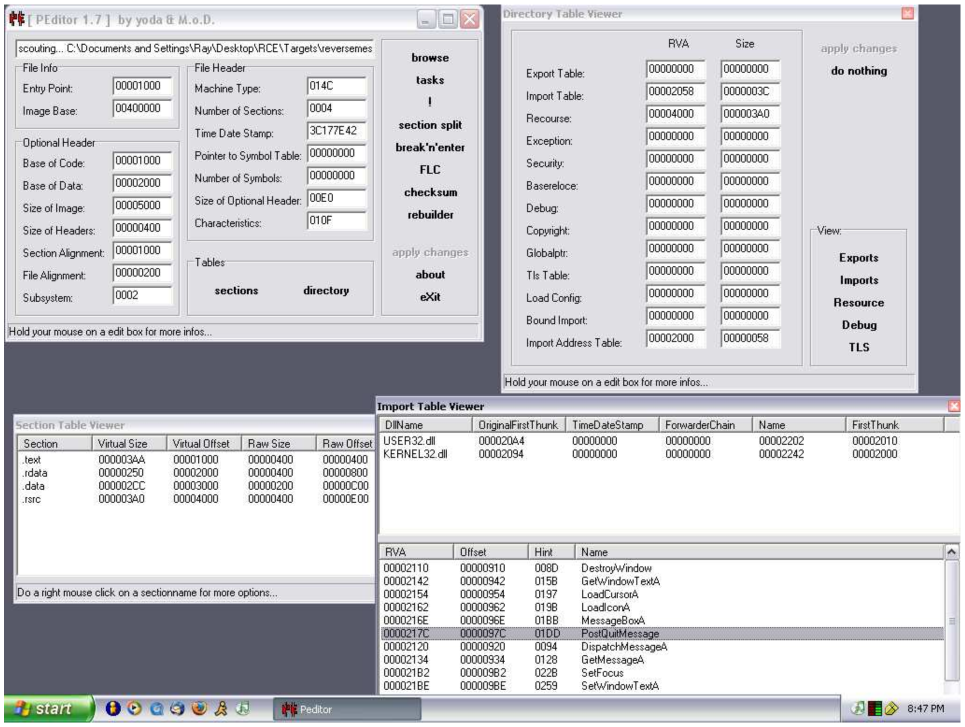


Fig. 16. Execution of a PE Editor

7.1 General PE Editors

When manual modifications are necessary, PE Editors are useful for displaying and modifying PE header and import table information. Many PE editors are available, including ProcDump, LordPE, and PEditor. PEditor [Danehkar 2004] supports the editing of PE headers, section tables, directory tables, and the imports and exports. These features are common to most PE Editors. PEditor also includes support for section splitting and checksum repairing, as well as an automatic rebuilder, which will attempt to repair and optimize altered PE headers automatically. Figure 17, a screenshot of PEditor, shows the amount of information a simple PE editor is capable of displaying. The top left displays the base PE header, the top right shows the directory table, the bottom left shows the section table, and the bottom right shows the imports.



scouting... C:\Documents and Settings\Ray\Desktop\RCE\Targets\reversemes

File Info

Entry Point:	00001000
Image Base:	00400000

Optional Header

Base of Code:	00001000
Base of Data:	00002000
Size of Image:	00005000
Size of Headers:	00000400
Section Alignment:	00001000
File Alignment:	00000200
Subsystem:	0002

File Header

Machine Type:	014C
Number of Sections:	0004
Time Date Stamp:	3C177E42
Pointer to Symbol Table:	00000000
Number of Symbols:	00000000
Size of Optional Header:	00E0
Characteristics:	010F

Hold your mouse on a edit box for more infos...

	RVA	Size
Export Table:	00000000	00000000
Import Table:	00002058	0000003C
Recourse:	00004000	000003A0
Exception:	00000000	00000000
Security:	00000000	00000000
Basereloce:	00000000	00000000
Debug:	00000000	00000000
Copyright:	00000000	00000000
Globalptr:	00000000	00000000
Tls Table:	00000000	00000000
Load Config:	00000000	00000000
Bound Import:	00000000	00000000
Import Address Table:	00002000	00000058

Hold your mouse on a edit box for more infos...

Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset
.text	000003AA	00001000	00000400	00000400
.rdata	00000250	00002000	00000400	00000800
.data	000002CC	00003000	00000200	00000C00
.rsrc	000003A0	00004000	00000400	00000E00

Do a right mouse click on a sectionname for more options...

DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
USER32.dll	000020A4	00000000	00000000	00002202	00002010
KERNEL32.dll	00002094	00000000	00000000	00002242	00002000

RVA	Offset	Hint	Name
00002110	00000910	008D	DestroyWindow
00002142	00000942	015B	GetWindowTextA
00002154	00000954	0197	LoadCursorA
00002162	00000962	019B	LoadIconA
0000216E	0000096E	01BB	MessageBoxA
0000217C	0000097C	01DD	PostQuitMessage
00002120	00000920	0094	DispatchMessageA
00002134	00000934	0128	GetMessageA
000021B2	000009B2	022B	SetFocus
000021BE	000009BE	0259	SetWindowTextA

Fig. 17. Screenshot of PEditor

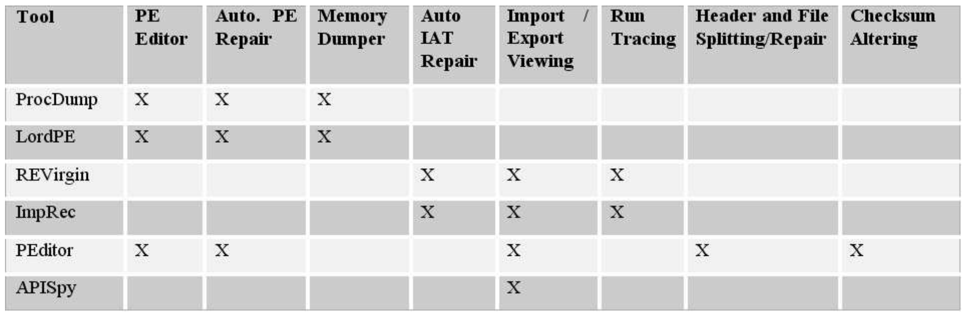
Procdump [G-RoM et al. 2000] and LordPE [Bzdok 2002] contain most of the features of PEditor, but do not support import, export, or resource viewing or editing. They stand out, however, because of their added features, including a memory dumper capable of dumping an entire process, or a user-defined section of a process. LordPE also supports merging split binaries, while ProcDump includes a "universal" unpacker.

7.2 Tools Specific to Import Address Tables

OllyDump [Gigapede 2004], a memory dumper plug-in for OllyDbg, attempts to fix the PE header and import table when dumping a program, but fails when the import table has been altered manually. As a result, additional effort is needed to restore the import table and make the program executable again, and a basic PE Editor is often inadequate. Two tools in particular focus on restoring the import tables, REVirgin [Tsehp 2002] and ImpRec [Mackt 2003]. Both programs attach to a running process and monitor it as it executes, attempting to discover the RVA of each imported function, utilizing multiple passes and run tracing. They also identify any unresolved imports, and support saving the discovered table to a dumped executable. ImpRec's functionality is automated, while REVirgin allows for user intervention when automation fails.

APISpy [Evseenko 1999] is a tool specific to identifying imported functions, and the locations where they are called. APISpy allows for the viewing of imports statically, and also allows a program to be executed, so the imports can be traced. The trace of the program displays the offsets where each of the imported functions were called. APISpy, however, is not a PE Editor or an IAT editor. It only allows the viewing of imported functions. APISpy also differs from the other tools we reviewed in this section because it is the only shareware utility (the others are freeware).

Table VII compares the following features of PE Editors: The ability to easily edit PE headers (PE Editor), tools for automatic repair of PE headers (Auto PE Repair), a function that allows an executable from memory to be saved to the disk (Memory Dumper), automatic import address table rebuilding (Auto IAT Repair), import and export table viewing (Import/Export Viewing), tracing the execution of a binary (Run Tracing), splitting and merging PE headers and files (Header and File Splitting/Repair), and altering checksums manually (Checksum Altering).



Tool	PE Editor	Auto. PE Repair	Memory Dumper	Auto IAT Repair	Import / Export Viewing	Run Tracing	Header and File Splitting/Repair	Checksum Altering
ProcDump	X	X	X					
LordPE	X	X	X					
REVirgin				X	X	X		
ImpRec				X	X	X		
PEditor	X	X			X		X	X
APISpy					X			

Table VII. Comparison of Major Features of PE Editors

8. CONCLUSIONS AND SHORTCOMINGS OF THE TOOLS

Our survey of reverse engineering tools for the Microsoft Windows environment comprises four categories: hex editors, disassemblers/debuggers, decompilers, and related technologies. The related technologies were then further divided into Code obfuscators, PE Editors and Unpackers. We reviewed several popular tools, determined which tools in each category appear best for specific applications, and presented their main features along with their applicability and usability. As our survey progressed, we found that some tools had noticeable shortcomings.

8.1 Hex Editors

All the hex editors we reviewed provide users with a method of manipulating the hexadecimal representation of a file. However, most hex editors appear to be lacking more advanced features such as RAM hex editing, data recovery, and disk utilities. The compare feature in some of the more basic hex editors such as HHD Hex Editors was also rather primitive.

8.2 Disassemblers/Debuggers

While IDA Pro features accurate analyses and disassembly, its debugging capabilities were limited to a simplistic debugger available as a menu option. OllyDbg, because it features both disassembly and debugging capabilities, is more useful. OllyDbg, however, has one major shortcoming: its inability to trace into `SYSENTER` calls. This command, introduced by the Intel Pentium II processor, was designed for making fast system calls. It is often invoked by Windows Structured Exception Handler, and thus employed by many advanced protection schemes. These protection schemes use the exception handler to execute necessary code, check if a program is being debugged, and clear the debug registers. When OllyDbg hits a `SYSENTER` command, however, it is unable to trace into it, and simply executes until the next exception. This is a major shortcoming of the debugger, and requires the user to break at `SYSENTER` commands, analyze the registers, and then determine where the target code of the `SYSENTER` command is, in order to trace into it. While SoftIce can trace into these commands, it is limited by the abundance of anti-SoftIce code that is constantly being developed to counter this mainstream debugger.

8.3 Decompilers

The decompilers we surveyed had numerous shortcomings. We were, however, satisfied with the operation of the Java decompilers, as they were able to decompile even obfuscated code with great accuracy. For other decompilers, especially the C decompilers, we found that the ability to decompile code was often limited to programs compiled for Microsoft DOS. We were also disappointed by the low accuracy and limited readability of the decompiled code. In most cases, understanding a decompiled executable was difficult and only practical for experienced programmers. We hope that as decompilers evolve, their accuracy will improve and their ability to produce better human-readable code will materialize.

REFERENCES

Backer Street Software 1997-2000. Reverse engineering compiler(rec). http://www.backerstreet.com/rec/rec.htm.

bartĈrackPl. 2000. *NFO*. http://protools.anticrack.de/files/packers/nfo.zip.

Bessonov, A. 2000. *HHD Hex Editor*. HHD Software. http://www.hhdsoftware.com/ hexeditor.html.

BreakPoint Software 2004. Hex workshop. http://www.bpsoft.com/.

Bruyere, S. 2003. *VBReFormer*. http://www.decompiler-vb.tk/.

Bzdok, D. 2002. *LordPE*. http://y0da.cjb.net/.

Cerven, P. 2002. *Crackproof Your Software*. No Starch Press.

CHAROLOIS, R. 2001. *Back to Java*. http://www.backtojava.org.

Cifuentes, C. 1991-1994a. *DCC*. http://www.itee.uq.edu.au/ cristina/dcc.html.

Cifuentes, C. 1994b. Reverse compilation techniques. Ph.D. thesis, Queensland University of Technology.

Compuware Corporation 2004. Softice. http://www.compuware.com/.

Cronos. 2001. *BORG*. http://homepage.ntlworld.com/cronos/.

DaFixer. 1999-2002. *DeDe*. http://www.balbaro.com/.

Danehkar, A. 2004. *PEditor*. http://freak2freak.cjb.net/.

Danehkar, A. and Bzdok, D. 2004. *Yoda's Crypter*. http://www.softpedia.com/progDownload/Yodas-Crypter-Download-24.html.

DataRescue. 2004. *IDA Pro*. http://www.datarescue.com/.

Dulek 2000. unnfo. http://protools.reverse-engineering.net/files/unpackers/unnfo.zip.

Evseenko, V. 1999. *APISpy*. http://www.matcode.com/apis32.htm.

Fleischmann, S. 1995-2004. *WinHex*. SF-Soft. http://www.x-ways.net/winhex/index-m.html.

Fuckar, V. 2003. *Debuggy*. http://web.vip.hr/inga.vip/.

G-RoM, Lorian, and Stone. 1998-2000. *ProcDump*. http://procdump32.cjb.net/.

Gabler, C. 2001. *Generic Unpacker for Windows (GUW)*. http://protools.anticrack.de/files/unpackers/guw32.zip.

Gigapede. 2004. *OllyDump*. http://ollydbg.win32asmcommunity.net/stuph/.

Hoenicke, J. 2004. *JODE*. http://jode.sourceforge.net.

Kaparo. 2004. *Programmers Tools*. http://protools.anticrack.de.

KnlSoft. 2002. *TRW*. http://www.knlsoft.com/.

Kouznetsov, P. 1997-2001. *JAD*. http://kpdus.tripod.com/jad.html.

Kumar, S. 2001-2003. *DISC*. http://www.debugmode.com/dcompile/disc.htm.

Mackt 2003. Import reconstructor. http://www.woodmann.com/crackz/Unpackers/Imprec16.zip.

Mead, I. 2004. *UltraEdit*. IDM Computer Solutions. http://www.ultraedit.com/.

Microsoft Corporation 2004. The msdn library. http://msdn.microsoft.com/library/default.asp.

Miecznikowski, J. 2004. *DAVA*. http://www.sable.mcgill.ca/soot/.

Miecznikowski, J. and Hendren, L. 2001. Decompiling java using staged encapsulation. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'01)* (Stuttgart, Italy).

Neshkov, A. 2004. *DJ*. http://members.fortunecity.com/neshkov/dj.html.

Peikari, C. and Chuvakin, A. 2004. *Security Warrior*. O'Reilly & Associates.

PreEmptive Solutions 2004. Dasho. http://www.preemptive.com/products/dasho.

SemanticDesigns 1995-2004. Source code obfuscators. http://www.semdesigns.com/Products/Obfuscators/index.html.

Silicon Realms Toolworks 1998-2004. Armadillo. http://www.siliconrealms.com/armadillo.shtml.

Snaker and Qwerton 2004. Peidentifier. http://peid.has.it/.

Solodovnikov, A. 2003. *AsProtect*. AsPack Software. http://www.aspack.com/.

TechnoLogismiki. 1996-2004. *Hackman Products*. http://www.technologismiki.com.

Tsehp 2002. Revirgin. http://tsehp.cjb.net/.

URSoftware. 1997. *W32DASM*. http://www.expage.com/page/w32dasm.

VanEmmerik, M. 2004. *Java Decompilers*. http://www.program-transformation.org/Transform/JavaDecompilers.

Yuschuk, O. 2000-2004. *OllyDbg*. http://home.t-online.de/home/Ollydbg.