51 Decompilation / EXE to C

"Wisdom is better than weapons of war."

Decompilation is the reverse of compilation. That is, we can get a C file from EXE file! The most important problem in converting back C file from EXE file is loss of variable names and loss of function names. Machine code won't store variable names. So it is not at all possible to get back the original C code.

51.1 Basic Idea

Since it is a reverse of compilation, we must analyze how a compiler works and the corresponding machine code for the functions like `printf( ), scanf( )` etc. In other words, we must find the 'signature' of each C functions and C statements.

51.2 DCC

51.2.1 Disclaimer

DCC is a decompiler written by **Cristina Cifuentes** and **Mike Van Emmerik** while at the Queensland University of Technology, Australia. Copyright is owned by **Cristina Cifuentes** and the Queensland University of Technology. DCC is merely a prototype tool and more work needs to be done in order to have a fully working decompiler.

Important Notice
I have received permission to use the article about DCC from the authors (Cristina Cifuentes and Mike Van Emmerik) with the condition of including the above disclaimer note. As Cristina Cifuentes and Mike Van Emmerik are not currently involving in decompililation, it seems they don't like to receive any request or correspondence regarding their decompilation work. So the reader is requested **not** to disturb them.

51.2.2 Notice

Decompilation is a technique that allows you to recover lost source code. It is also needed in some cases for computer security, interoperability and error correction. dcc, and any decompiler in general, should not be used for "cracking" other programs, as programs are protected by copyright. Cracking of programs is not only illegal but it rides on other's creative effort.

51.2.3 DCC Facts

The dcc decompiler decompiles .exe files from the (i386, DOS) platform to C programs. The final C program contains assembler code for any subroutines that are not possible to be decompiled at a higher level than assembler.

The analysis performed by dcc is based on traditional compiler optimization techniques and graph theory. The former is capable of eliminating registers and intermediate instructions to reconstruct high-level statements; the later is capable of determining the control structures in each subroutine.

Please note that at present, only C source is produced; dcc cannot (as yet) produce C++ source.

The structure of a decompiler resembles that of a compiler: a front-, middle-, and back-end which perform separate tasks. The front-end is a machine-language dependent module that reads in machine code for a particular machine and transforms it into an intermediate, machine-independent representation of the program. The middle-end (aka the Universal Decompiling Machine or UDM) is a machine and language independent module that performs the core of the decompiling analysis: data flow and control flow analysis. Finally, the back-end is high-level language dependent and generates code for the program (C in the case of dcc).

In practice, several programs are used with the decompiler to create the high-level program. These programs aid in the detection of compiler and library signatures, hence augmenting the readability of programs and eliminating compiler start-up and library routines from the decompilation analysis.

51.2.4 Example of Decompilation

We illustrate the decompilation of a fibonacci program (see Figure 4). Figure 1 illustrates the relevant machine code of this binary. No library or compiler start up code is included. Figure 2 presents the disassembly of the binary program. All calls to library routines were detected by dccSign (the signature matcher), and thus not included in the analysis. Figure 3 is the final output from dcc. This C program can be compared with the original C program in Figure 4.

```
      55 8B EC 83 EC 04 56 57 1E B8 94 00 50 9A
0E 00 3C 17 59 59 16 8D 46 FC 50 1E B8 B1 00 50
9A 07 00 F0 17 83 C4 08 BE 01 00 EB 3B 1E B8 B4
00 50 9A 0E 00 3C 17 59 59 16 8D 46 FE 50 1E B8
C3 00 50 9A 07 00 F0 17 83 C4 08 FF 76 FE 9A 7C
00 3B 16 59 8B F8 57 FF 76 FE 1E B8 C6 00 50 9A
0E 00 3C 17 83 C4 08 46 3B 76 FC 7E C0 33 C0 50
9A 0A 00 49 16 59 5F 5E 8B E5 5D CB 55 8B EC 56
8B 76 06 83 FE 02 7E 1E 8B C6 48 50 0E E8 EC FF
59 50 8B C6 05 FE FF 50 0E E8 E0 FF 59 8B D0 58
03 C2 EB 07 EB 05 B8 01 00 EB 00 5E 5D CB
```

Figure 1 - Machine Code for Fibonacci.exe

```
                proc_1  PROC  FAR
000 00053C 55                   PUSH            bp
001 00053D 8BEC                 MOV             bp, sp
002 00053F 56                   PUSH            si
003 000540 8B7606               MOV             si, [bp+6]
004 000543 83FE02               CMP             si, 2
005 000546 7E1E                 JLE             L1
006 000548 8BC6                 MOV             ax, si
007 00054A 48                   DEC             ax
008 00054B 50                   PUSH            ax
009 00054C 0E                   PUSH            cs
010 00054D E8ECFF               CALL  near ptr proc_1
011 000550 59                   POP             cx
012 000551 50                   PUSH            ax
013 000552 8BC6                 MOV             ax, si
014 000554 05FEFF               ADD             ax, 0FFFEh
015 000557 50                   PUSH            ax
016 000558 0E                   PUSH            cs
017 000559 E8E0FF               CALL  near ptr proc_1
018 00055C 59                   POP             cx
019 00055D 8BD0                 MOV             dx, ax
020 00055F 58                   POP             ax
021 000560 03C2                 ADD             ax, dx
023 00056B 5E             L2:   POP             si
024 00056C 5D                   POP             bp
025 00056D CB                   RETF
026 000566 B80100         L1:   MOV             ax, 1
027 000569 EB00                 JMP             L2
                proc_1  ENDP

                main  PROC  FAR
000 0004C2 55                   PUSH            bp
001 0004C3 8BEC                 MOV             bp, sp
002 0004C5 83EC04               SUB             sp, 4
003 0004C8 56                   PUSH            si
004 0004C9 57                   PUSH            di
005 0004CA 1E                   PUSH            ds
006 0004CB B89400               MOV             ax, 94h
007 0004CE 50                   PUSH            ax
008 0004CF 9A0E004D01           CALL   far ptr printf
009 0004D4 59                   POP             cx
010 0004D5 59                   POP             cx
011 0004D6 16                   PUSH            ss
012 0004D7 8D46FC               LEA             ax, [bp-4]
013 0004DA 50                   PUSH            ax
014 0004DB 1E                   PUSH            ds
015 0004DC B8B100               MOV             ax, 0B1h
```

```
016 0004DF 50                   PUSH            ax
017 0004E0 9A07000102           CALL   far ptr scanf
018 0004E5 83C408               ADD             sp, 8
019 0004E8 BE0100               MOV             si, 1
021 000528 3B76FC         L3:   CMP             si, [bp-4]
022 00052B 7EC0                 JLE             L4
023 00052D 33C0                 XOR             ax, ax
024 00052F 50                   PUSH            ax
025 000530 9A0A005A00           CALL   far ptr exit
026 000535 59                   POP             cx
027 000536 5F                   POP             di
028 000537 5E                   POP             si
029 000538 8BE5                 MOV             sp, bp
030 00053A 5D                   POP             bp
031 00053B CB                   RETF
032 0004ED 1E             L4:   PUSH            ds
033 0004EE B8B400               MOV             ax, 0B4h
034 0004F1 50                   PUSH            ax
035 0004F2 9A0E004D01           CALL   far ptr printf
036 0004F7 59                   POP             cx
037 0004F8 59                   POP             cx
038 0004F9 16                   PUSH            ss
039 0004FA 8D46FE               LEA             ax, [bp-2]
040 0004FD 50                   PUSH            ax
041 0004FE 1E                   PUSH            ds
042 0004FF B8C300               MOV             ax, 0C3h
043 000502 50                   PUSH            ax
044 000503 9A07000102           CALL   far ptr scanf
045 000508 83C408               ADD             sp, 8
046 00050B FF76FE               PUSH  word ptr [bp-2]
047 00050E 9A7C004C00           CALL   far ptr proc_1
048 000513 59                   POP             cx
049 000514 8BF8                 MOV             di, ax
050 000516 57                   PUSH            di
051 000517 FF76FE               PUSH  word ptr [bp-2]
052 00051A 1E                   PUSH            ds
053 00051B B8C600               MOV             ax, 0C6h
054 00051E 50                   PUSH            ax
055 00051F 9A0E004D01           CALL   far ptr printf
056 000524 83C408               ADD             sp, 8
057 000527 46                   INC             si
058                             JMP             L3          ;Synthetic inst
                main  ENDP
```

Figure 2 - Code produced by the Disassembler

```
/*
 * Input file  : fibo.exe
 * File type   : EXE
 */
int proc_1 (int arg0)
/* Takes 2 bytes of parameters.
 * High-level language prologue code.
 * C calling convention.
 */
{
int loc1;
int loc2; /* ax */

    loc1 = arg0;
    if (loc1 > 2) {
        loc2 = (proc_1 ((loc1 - 1)) + proc_1 ((loc1 + 0xFFFE)));
    }
    else {
        loc2 = 1;
    }
    return (loc2);
}

void main ( )
/* Takes no parameters.
 * High-level language prologue code.
 */
{
int loc1;
int loc2;
int loc3;
int loc4;

    printf ("Input number of iterations: ");
    scanf ("%d", &loc1);
    loc3 = 1;
    while ((loc3 <= loc1)) {
        printf ("Input number: ");
        scanf ("%d", &loc2);
        loc4 = proc_1 (loc2);
        printf ("fibonacci(%d) = %u\n", loc2, loc4);
        loc3 = (loc3 + 1);
    } /* end of while */
    exit (0);
}
```

Figure 3 - Code produced by dcc in C

```
#include <stdio.h>

int main( )
{ int i, numtimes, number;
  unsigned value, fib();

   printf("Input number of iterations: ");
   scanf ("%d", &numtimes);
   for (i = 1; i <= numtimes; i++)
   {
      printf ("Input number: ");
      scanf ("%d", &number);
      value = fib(number);
      printf("fibonacci(%d) = %u\n", number, value);
   }
   exit(0);
}

unsigned fib(x)        /* compute fibonacci number recursively */
int x;
{
   if (x > 2)
      return (fib(x - 1) + fib(x - 2));
   else
      return (1);
}
```

Figure 4 – Initial / Original C Program