UNIVERSAL DISASSEMBLER

Aleš Smrčka, Master Degree Programme (3)
Dept. of Intelligent Systems, FIT, BUT
e-mail: xsmrck01@stud.fit.vutbr.cz

Supervised by: Dr. Petr Peringer

ABSTRACT

The project deals with problem how to disassemble the binary code into symbolic instructions. The purpose of this work is to create the object oriented design of a program and to implement the design in the C++ language. The program will be used to disassemble many different executable file formats with binary code using various instruction sets.

1 INTRODUCTION

By compilation of a program written in programming language we get a binary code for processor. Sometimes it is a binary code to be processed by virtual processors (e.g. Java Virtual Machine). Reverse program translation is the translation of binary code into a code readable by a human. The human readable format could be symbolic instruction language (the lowest level of reverse program translation) or some kind of programming language (the highest level of reverse translation). The program used for reverse translation of binary code to language of symbolic instruction is called disassembler. This project deals with disassembling of different binary file formats. The program can naturally handle different instruction sets (i.e. different processor types).

2 CODE ANALYSIS

Compiled program is saved into executable file. There are several different formats of executable files. Some of them are usable only for some operating systems. Generally in every executable file there are several sections. Some sections contain instructions, some contain data, constant data etc.

In disassembler it is important to distinguish two types of sections. The first type is data section, the second is executable section — section containing instructions for processor. Data section is disassembled into simple output of its content, which can be either in hexadecimal text format or in binary format.

3 DESIGN OF DISASSEMBLER

Disassembler consists of three main parts. The first part solves the access to the sections of input file, the second part is a symbol table and the last one deals with instruction decoding according to instruction sets. We used design patterns [1] and UML (Unified Modeling Language [2]) during object-oriented design of this program.

3.1 SYMBOL TABLE

Commented text and list of addresses in program are maintained in symbol table. This table maps addresses into lables. The content of the symbol table is read from special section of executable file. It may be also read from external user defined text file which contains comments of the addresses.

3.2 INSTRUCTION SET DECODER

For reverse translation it is necessary to know instruction set which is needed for decoding of instructions. This set is composed of many tables and decoding variables.

Decoding process starts by decoding the first byte of executable section. Every byte of sections is used as index into the table. This way we gain an decoding expression. This expression may contain references to other tables indexed by next bytes and references to values of the decoding variables. Instruction set decoder proccess this expression according to the next bytes of executable section and the result of this expression will be a symbolic instruction. For description of decoding variables, tables and their decoding expressions I was created a special language which will be described in the next paragraph.

3.3 INSTRUCTION SET DESCRIPTION LANGUAGE

Instruction sets are described in text files. The instruction set language is described by following rewriting rules. `ident` is an identifier following the definition of identifier in C language. `char` is list of any character except comma, quotation marks and these control characters: `$`,`{`,`}`,`(`,`)`. `number` is a number in decimal or hexadecimal format, `letter` is one letter of the English alphabet (a–z).

```
      MAIN  →  VARIABLE MAIN
      MAIN  →  TABLE MAIN
      MAIN  →  $

  VARIABLE  →  ident = “ char ” ;

     TABLE  →  = { TABLEITEMS }
TABLEITEMS  →  TABLEITEM ; TABLEITEMS
TABLEITEMS  →  e
 TABLEITEM  →  number , number , TEXT , LENGTH
```

```
     TEXT  →  “ ITEMTEXT ”
 ITEMTEXT  →  $ { ITEMTEXT } ITEMTEXT
 ITEMTEXT  →  char ITEMTEXT
 ITEMTEXT  →  $ ( ITEMTEXT , INDEX ) ITEMTEXT
 ITEMTEXT  →  e

   LENGTH  →  number LENGTHS
   LENGTH  →  letter LENGTHS
  LENGTHS  →  + LENGTH
  LENGTHS  →  e
```

4 IMPLEMENTATION

The project is written in C++ language (using standard template library — stl). The functionality is based on plug-ins and external definition of an instruction set. There is already implementation of plug-in for reading the unix ELF format (binary executable format) and the plug-in for reading pure instruction stream (e.g. file containing only content of one executable section). The project is published on the freshmeat.net site (http://freshmeat.net/projects/uda).

5 CONCLUSION

There are available many disassemblers, but these do not correspond with our requirements. Free software is simple and commercial software is expensive. The best known and high-quality product is IDA Pro ([3]) which inspired me in many ways. The goal of universal disassembler is to have the best qualities of all so far known projects. It’s usage is universal, it is platform independent, it is for free and protected by the GNU GPL license [4]. Thanks to standard C++ language and POSIX standardization it is possible to compile and subsequently use this disassembler on several operating systems.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Software Development, Productivity Award 1994. ISBN:0-201-63361-2

[2] *Unified Modeling Language*, URL: http://www.uml.org/

[3] *IDA Pro — The Interactive Disassembler*, URL: http://www.datarescue.com/idabase/

[4] *The GNU General Public License,* URL: http://www.gnu.org/licenses/licenses.html