Decompilation of Binary Programs by Cristina Cifuentes and John Gough

Rob Meadows

1 Introduction

A decompiler is a program that takes as input an executable program (that was compiled in some high level language), and reproduces a high level language representation of that program. This paper by Cristina Cifuentes and John Gough discusses a method of decompilation, including the structure and modules of a decompiler, the analyses done during decompilation, and an example of the particular decompiler named dcc that these authors developed.

This paper first discusses some background information about decompilers such as the problems with, uses for, and limitations of the current decompilers available. It is often difficult to separate the data, the author's code, and the compiler's code out of an executable. Some decompilers have added limitations which help simplify this process, such as restricting the input to assembly code. However, a general decompiler is more desirable and can be beneficial in the areas of code maintenance and code security.

The decompiler described in this paper has three main modules: The front-end, the Universal Decompiling Machine (UDM), and the back-end. The front-end loads the executable program into memory and parses it. It performs some semantic analysis such as idiom analysis and type propagation and constructs the control flow graph. The output of the front-end is an intermediate low level code. The UDM performs the data flow analysis on the condition codes and the registers. It then structures the control flow graph into higher level control structures. The output of the UDM is an intermediate high level code. Finally, the back-end takes the intermediate high level code and the control flow graph, and produces a program in some high level language.

The specific decompiler discussed in this paper is named dcc. This decompiler takes as input a binary program from the DOS environment (exe or

com file), and produces a C program. The dcc decompiler looks at compiler signatures in order to determine what code was produced by the compiler (and is thus not decompiled). Dcc also replaces known library code with library routine calls, making the decompiled program more readable.

2 General Decompiler Issues

Why do we need a decompiler? Maintenance and Security! A decompiler is very useful in the recovery of lost source code. Imagine writing a new word-processor that's sure to take over the Microsoft Word market. You compile it and give it to a few friends to try. The next day your computer somehow catches fire (Microsoft has good lawyers...). It would be nice to get the executable from your friend and simply decompile it to get your valuable source code back. Decompilation is also useful in migrating applications to new hardware platforms. Machine code is not generally transportable, but a high level language is (in theory). You can simply recompile the high level code on any new hardware platform.

Another benefit of decompilation is the ability to translate obsolete languages into modern languages. It is difficult to maintain code that was written in Algol or Fortran when new Computer Scientists are being taught Java and C++. A decompiler may be able to reconstruct a program compiled from an old language into a high level modern language. Additionally, decompilation can help structure old "spaghetti code" into a more structured high level language. Finally, decompilation can serve as a debugging tool for existing binary programs. Often times, the authors of software are too busy or incapable of fixing bugs that you may need fixed. Decompilation can help you track down a bug and fix the defect if possible.

Security is becoming a hot issue in computing today, and a decompiler can be a valuable tool in this area. When new software is obtained, it can be decompiled and checked for malicious code before it is executed for the first time. Additionally, when the output of some compiler is not trusted, the executable can be decompiled and compared to the original high level code. Hopefully it's obvious that a decompiler is a valuable resource, but what is involved in writing one?

One of the problems that decompiler writers must face is separating the instructions from the data. In high level languages, instructions and variables are two separate things. However, in the Von Neumann architecture, data and instructions both appear as a simple series of bits. For example, an indexed jump table containing data can often be found in the middle of

a series of machine instructions.

Once the data can be separated from the instructions, the decompiler writers must then face the problem of separating the instructions that were added by the compiler or linker from the instructions that were actually translated from the author's original high level code. Most compilers add assembly code that prepares the execution environment. Additionally, most operating systems cannot share compiled library routines, so these routines (often written in a low level language) must be put directly into the compiled program. A lot of this assembly code is not even translatable into high level code or is useless at a high level. For example, the Hello World program compiled in C has 23 procedures, and 40 procedures in Pascal. A decompiler must have a method to filter out this extra information and produce only the high level code that was produced by the programmer.

Because these difficulties in writing decompilers are non-trivial, many decompiler implementations have added limitations to simplify them. Some decompilers require input in the form of an assembly file or an object file. These types of input contain more information than a binary executable such as data segments, type declarations, subroutine names, and symbol tables. This extra information helps eliminate the problem of data/instruction separation. Other decompilers can only produce a simplified subset of some high level language or require the specifications of the original compiler. However, most compiler manufacturers will not disclose this information, even if it does exist. These types of limited decompilers can have some use, but predictably, their uses are limited. The authors of this paper describe the structure and method of a better, more general decompiler.

3 Decompiler Structure

The overall structure of the dcc decompiler is a series of three modules. The front-end is machine specific, as it produces a low level language from the binary executable. The back-end produces an output in a particular high level language and is thus language specific. The Universal Decompiling Machine is both machine and language independent, as it translates an intermediate low level language (produced by the front end) to an intermediate high level language (used by the back-end). The details of each module will be examined further.

The front-end loads the binary program into working memory and parses it. This parsing is accomplished by starting the parser at the entry point, and following the instructions sequentially. When a fork in the instruction

path is encountered (at conditionals, branches, etc), the parser recursively explores each path. A low level intermediate code that resembles assembly is built during this parsing stage. The front-end also performs a limited amount of semantic analysis - namely, idiom analysis and type propagation. Idioms are known sets of instructions that perform certain simple tasks. An example of this is the negation of a long variable. The three instructions that perform this operation would be replaced by the one intermediate instruction in this phase. After such an idiom is recognized, the type information can then be propagated. For example, if this long variable is being stored in two registers, the references to these registers must be changed to reference a long variable everywhere else. The final responsibility of the front-end is to construct the control flow graph. After the graph is constructed, some simple optimizations can be made such as removing intermediate branches. For example, if a jump statement jumps to another jump statement, the address of the first jump can be changed and the second jump removed.

Once the intermediate low level code and control flow graph have been produced, the Universal Decompiling Machine can begin it's work. Since high level languages do not contain the concept of condition codes and registers, these need to be eliminated through the introduction of expressions. There are two types of condition codes that must be dealt with: those that are likely to be produced by a compiler and those that are likely to be produced by assembly code. If the second type of condition code is encountered, that portion of code need not be decompiled, as it was not produced from any high level language. It was most likely added by the compiler. A use/definition analysis is then performed on the valid condition codes. This analysis determines which instructions set condition codes that later instructions use. Two instructions can then be combined into a single expression. For example, a compare may set the zero condition code that is later used in a branch. These two instructions can be combined into a high level conditional expression such as: if (a¿b) then branch.

The elimination of registers is accomplished by replacing the registers with local variables. This reduction also eliminates many intermediate instructions that use temporary registers. To accomplish this, a definition/use analysis is performed to determine how many uses of a register there are for a particular definition of that register. If a register is only used once for a given definition, it can usually be eliminated since it is a temporary variable. Howeverm Some assembly code cannot be replaced by a high level instruction that eliminates the temporary registers and still accomplishes the same task. Like the assembly generated condition codes, this code is not produced by the compiler and is thus not decompiled any further.

The final task of the UDM is to add high level control structures to the control flow graph. This is accomplished through further control flow analysis. The graph is structured into a set of generic high-level control structures (if-then-else, case, while, repeat, etc). If one of these constructs cannot be used, a goto is used. This algorithm produces the graph with the minimal number of goto's. The CFG can be further simplified by adding short circuited evaluation. For example, `[if(a>b) then if (c>d)] becomes [if (a>b && c>d)]`.

The last phase of the decompilation process is the back-end module. This module is responsible for translating the intermediate high level language into a some particular high level language such as C or Pascal. Names are chosen for variables (such as var1 and proc2, as the original names are lost in compilation). Global variables are defined, and each procedure is translated into high level code. The control flow graph may be further reconstructed, depending on what conditional constructs the language supports. For example, if case statements are not supported, they must be translated into nested if/elses. The final output is a complete program in a high level language.

4 The DCC decompiler

The DCC decompiler was developed by the authors using the decompiler structure detailed in the paper. This decompiler reads in .com and .exe files from an x86 environment and produces C code. Since so many additional procedures, library code, and compiler specific code is added during compilation, a method was developed to simplify or eliminate this extra code. Most compilers produce particular sets of instructions that can be thought of as the signature of that compiler. Once the compiler signature is determined, the known compiler code can then be ignored. Additionally, any known library code that is encountered is replaced with the library routine call (for example, printf()). The result is an output in C that looks very much like the original high level code, minus some variable and procedure names.

5 Conclusion

In conclusion, this research paper was very well written and understandable. It presented a good overview of decompilers along with many examples to help make the difficult concepts more concrete. The structure of the decompiler is presented in detail, and the methods used in each module are clearly

explained. The dcc decompiler is introduced, and detailed examples of its use are presented. It is clear that the process of decompilation is not trivial, but it is an exciting and interesting topic with cool applications and a lot of room for further research.