SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(7), 811–829 (JULY 1995)

Decompilation of Binary Programs

CRISTINA CIFUENTES* AND K. JOHN GOUGH
(email: cifuente@fit.qut.edu.au gough@fit.qut.edu.au)
School of Computing Science, Queensland University of Technology, GPO Box 2434, Brisbane QLD 4001, Australia

SUMMARY

The structure of a decompiler is presented, along with a thorough description of the different modules that form part of a decompiler, and the type of analyses that are performed on the machine code to regenerate high-level language code. The phases of the decompiler have been grouped into three main modules: front-end, universal decompiling machine, and back-end. The front-end is a machine-dependent module that performs the loading, parsing and semantic analysis of the input program, as well as generating an intermediate representation of the program. The universal decompiling machine is a machine- and language-independent module that performs data and control flow analysis of the program based on the intermediate representation, and the program's control flow graph. The back-end is a language-dependent module that deals with the details of the target high-level language.

In order to increase the readability of the generated programs, a decompiling system has been implemented which integrates a decompiler, *dcc*, and an automatic signature generator, *dccSign*. Signatures for libraries and compilers are stored in a database that is read by the decompiler; thus, the generated programs can make use of known library names, such as `WriteLn()` and `printf()`.

***dcc* is a decompiler for the Intel 80286 architecture and the DOS operating system. *dcc* takes as input binary programs from a DOS environment and generates C programs as output. Sample code produced by this decompiler is given.**

KEY WORDS: decompiler; reverse compiler; compiler signature; library signature; i80286; C language

INTRODUCTION

A decompiler, or reverse compiler, is a program that attempts to perform the inverse process of the compiler: given an executable program compiled in any high-level language, the aim is to produce a high-level language program that performs the same function as the executable program. Thus, the input is machine dependent, and the output is language dependent.

Several practical problems are faced when writing a decompiler. The main problem derives from the representation of data and instructions in the Von Neumann architecture: they are indistinguishable. Thus, data can be located in between instructions, such as many implementations of indexed jump (`case`) tables. This representation and self-modifying code practices makes it hard to decompile a binary program.

Another problem is the great number of subroutines introduced by the compiler and the linker. The compiler will always include start-up subroutines that set up its environment,

* Present address: Department of Computer Science, University of Tasmania, GPO Box 252C, Hobart TAS 7001, Australia. (Email: C.N.Cifuentes@cs.utas.edu.au)

CCC 0038–0644/95/070811–19

Received April 1994
Revised January 1995

and runtime support routines whenever required. These routines are normally written in assembler and in most cases are untranslatable into a higher-level representation. Also, most operating systems do not provide a mechanism to share libraries; consequently, binary programs are self-contained and library routines are bound into each binary image. Library routines are either written in the language the compiler was written in, or in assembler. This means that a binary program contains not only the routines written by the programmer, but a great number of other routines linked in by the linker. As an example of the amount of extra subroutines available in a binary program, a 'hello world' program compiled in C generates 23 different procedures. The same program compiled in Pascal generates more than 40 procedures. All we are really interested in is the one procedure, `main`.

Despite the above-mentioned limitations, there are several uses for a decompiler, including two major software areas: maintenance of code and software security. From a maintenance point of view, a decompiler is an aid in the recovery of lost source code, the migration of applications to a new hardware platform, the translation of code written in an obsolete language into a newer language, the structuring of old code written in an unstructured way (e.g. 'spaghetti' code), and a debugger tool that helps in finding and correcting bugs in an existing binary program. From a security point of view, a binary program can be checked for the existence of malicious code (e.g. viruses) before it is run for the first time on a computer, in safety-critical systems where the compiler is not trusted, the binary program is validated to do exactly what the original high-level language program intended to do, and thus, the output of the compiler can be verified in this way.

Different attempts at writing decompilers have been made in the last 20 years. Due to the amount of information lost in the compilation process, to be able to regenerate high-level language (HLL) code, all of these experimental decompilers have limitations in one way or another, including decompilation of assembly files[1,2,3,4,5] or object files with or without symbolic debugging information,[6,7] simplified high-level language,[1] and the requirement of the compiler's specification.[8,9] Assembly programs have helpful data information in the form of symbolic text, such as data segments, data and type declarations, subroutine names, subroutine entry point, and subroutine exit statement. All this information can be collected in a symbol table and the decompiler would not need to address the problem of separating data from instructions, or the naming of variables and subroutines. Object files with debugging information contain the program's symbol table as constructed by the compiler. Given the symbol table, it is easy to determine which memory locations are instructions, as there is a certainty on which memory locations represent data. In general, object files contain more information than binary files. Finally, the requirement to have access to the compiler's specifications is impractical, as these specifications are not normally disclosed by compiler manufacturers, or do not even exist.

Our decompiler, *dcc*, differs from previous decompilation projects in several ways; it analyses binary programs rather than assembler or object files, performs idiom* analysis to capture the essence of a sequence of instructions with a special meaning, performs data flow analysis on registers and condition codes to eliminate them, and structures the program's control flow graph into a generic set of high-level structures that can be accommodated into different high-level languages, eliminating as much as possible the use of the `goto` statement.

The rest of this paper is structured in the following way: a thorough description of the structure of a decompiler, followed by the description of our implementation of an

* An idiom is a sequence of instruction that forms a logical entity and has a meaning that cannot be derived by considering the primary meanings of the individual instructions

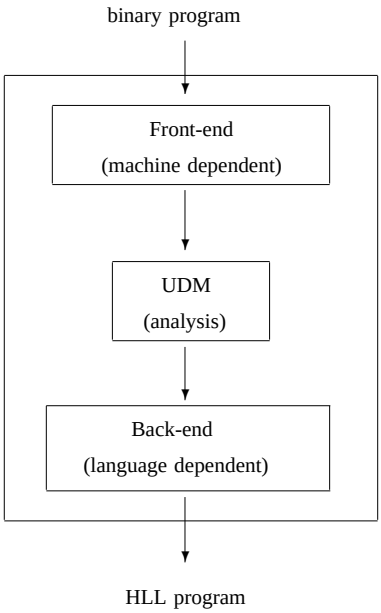

Figure 1. Decompiler modules

automatic decompiling system, and conclusions. The paper is followed by the definitions of graph theoretical concepts used throughout the paper (Appendix I), and sample output from different phases of the decompilation of a program (Appendix II).

THE DECOMPILER STRUCTURE

A decompiler can be structured in a similar way to a compiler, that is, a series of modules that deal with machine- or language-dependent features. Three main modules are required: a machine-dependent module that reads in the program, loads it into virtual memory and parses it (the front-end), a machine- and language-independent module that analyses the program in memory (the universal decompiling machine), and a language-dependent module that writes formatted output for the target language (the back-end) (see Figure 1). This modular representation makes it easier to write decompilers for different machine/target language pairs, by writing different front-ends for different machines, and different back-ends for different target languages. This result is true in theory, but in practical applications is always limited by the generality of the intermediate language used.

The front-end

The front-end module deals with machine-dependent features and produces a machine-independent representation. It takes as input a binary program for a specific machine, loads it into virtual memory, parses it, and produces an intermediate representation of the program, as well as the program's control flow graph (see Figure 2).

The *parser* disassembles code starting at the entry point given by the *loader*, and follows

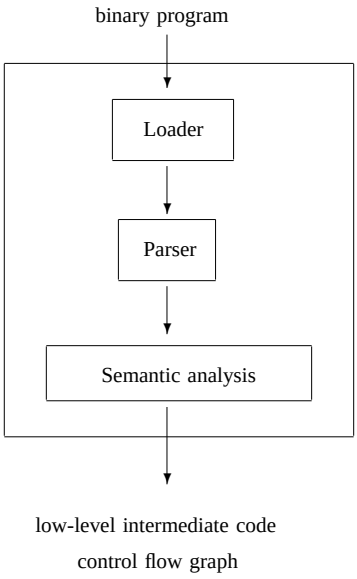

Figure 2. Front-end phases

the instructions sequentially until a change in flow of control is met. Flow of control is changed due to a conditional, unconditional or indexed branch, or a procedure call; in which case the target branch label(s) start new instruction paths to be disassembled. A path is finished by a return instruction or program termination. All instruction paths are followed in a recursive manner. Problems are introduced by machine instructions that use indexed or indirect addressing modes. To handle these, heuristic methods are implemented. For example, while disassembling code, the parser must check for sequences of instructions that represent a multiway branch (e.g. a `switch` statement), normally implemented as an index jump in a jump table.[10,11] Finally, the intermediate code is generated and the control flow graph is built.

Two levels of intermediate code are required; a low-level representation that resembles the assembler from the machine, and a higher-level representation that resembles statements from a high-level language. The initial level, or *low-level intermediate code*, is a simple m : 1 mapping of machine instructions to assembler mnemonics. Compound instructions (such as `rep movs`) are represented by a unique low-level intermediate intruction (e.g. `rep_movs`). The second level, or *high-level intermediate code*, is generated by the interprocedural data flow analysis, explained later, and maps n : 1 low-level to high-level instructions. The front-end generates a low-level representation.

The *semantic analysis* phase performs idiom analysis and type propagation. Idioms are replaced by an appropriate functionally equivalent intermediate instruction. For example, Figure 3 illustrates two different idioms: the one on the left-hand side is a negation of a long variable, represented in this case by registers `dx:ax`. The idiom on the right-hand side is the prologue code of a high-level language procedure. In this case, space for 6 bytes is being reserved on the stack for local variables. There are a number of different idioms

```
neg dx              push bp
neg ax              mov  bp, sp
sbb dx, 0           sub  bp, 6

                ⇓

neg dx:ax           enter 6,0
```

Figure 3. Sample idioms and their transformation

widely known in the compiler community, and the decompiler must code them in order to generate clearer high-level code.

Type information is propagated after the idioms have been recognized. For example, if a long local variable is found at stack offsets -1 to -4, all references to `[bp-2]` and `[bp-4]` must be merged into references to such a long variable, e.g. `[bp-2]:[bp-4]`. Other type information can be propagated in the same way, such as fields (offsets) of a record.

An optimization phase is performed on the control flow graph as well. Due to the nature of machine code instructions, the compiler might need to introduce intermediate branches in an executable program, because there is no machine instruction capable of branching more than a certain maximum distance in bytes (architecture dependent). An optimization pass over the control flow graph removes this redundancy, by replacing the target branch location of all conditional or unconditional jumps that branch to an unconditional jump (and any recursive branches in this format) with the final target basic block. While performing this process, some basic blocks are not going to be referenced any more, as they were used only for intermediate branches. These nodes must be eliminated from the graph as well.

The universal decompiling machine

The universal decompiling machine (UDM) is an intermediate module that is totally machine and language independent. It deals with flow graphs and the intermediate representation of the program and performs all the flow analysis the input program needs (see Figure 4).

Data flow analysis

The aim of the *data flow analysis* phase is to transform the low-level intermediate representation into a higher-level representation that resembles a HLL statement. It is therefore necessary to eliminate the concept of condition codes (or flags) and registers, as these concepts do not exist in high-level languages, and to introduce the concept of expressions, as these can be used in any HLL program. For this purpose, the technology of compiler optimization has been appropriated.

The first analysis is concerned with condition codes. Some condition codes are used only by hand-crafted assembly code instructions, and thus are not translatable to a high-level representation. Therefore, condition codes are classified in two groups: HLCC which is the set of condition codes that are likely to have been generated by a compiler (e.g. overflow, carry), and NHLCC which is the set of condition codes that are likely to have been generated by assembly code (e.g. trap, interrupt). The HLCC set is the one that can

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.