Two-Way-Compiler: Additional Data Saving for Generating the Original Source Code of a Binary Program

Dennis Obermann, Josef Börcsök

Department of Computer Architecture and System Programming
University Kassel
Wilhelmshöher Allee 71
34121 Kassel
obermann@uni-kassel.de

Abstract: The Two-Way-Compiler is an approach to show the equivalence between implemented source code and the generated binary program for safety-related software. A compiler which translates a source code into a binary program and restores the original source code out of the generated binary program exactly, like a decompiler, is described. Data that are required to build the original source code back again are especially examined in this paper. Some data are contained in the binary itself and other data lost during compilation. The lost data have to be collected and stored in the binary. With these additional data the decompiler can restore the binary program to the original source code.

1 Introduction

An implemented source code has to be translated by a compiler into a binary to get an executable program for safety-related systems. Frequently, the compiler is classified as not trustable. Therefore, intensive tests of the binary program against the requirements and the source code have to be performed. These tests are helpful to verify that the compiler does exactly what it should do. IEC 61508-3 mentions this in the development life cycle for software verification [IEC10]. A special test, which proves that the binary program corresponds exactly to the implemented source code, is not explicitly shown in this standard. But for the verification of safety-related software such a test is very important.

To verify safety-related software, it would be helpful to be able to translate the binary program back to its original representation. A simple comparison between the original source code and the decompiled source code will be possible.

A program that creates source code out of a binary program is called a decompiler. The first decompiler that translates binary programs from second generation computers to third generation computers was developed in the 1960s [Ha62]. In the following years, the techniques of decompiling were further developed and used to porting programs from one machine to another machine, to make documentation of assembler code, to rescue lost source code and to modify binary programs [Ci94]. Today, many decompilers are based on the phases of compilers and use identical techniques to analyze the input [Em07]. They analyze the binary program by graph theory, translate it into an intermediate representation and generate the source code. Control flow analysis and data flow analysis are very important [Ci94]. Human readability also plays a fundamental role [Ch10].

In some approaches the decompilers use assembler code [Sa66] or binary programs with debug informations as input. These inputs contain additional data that simplify the generation of the source code. There are symbolic informations about data segments, types, subroutine names, entry points and exit statements [Ci94]. Other decompilers are working on byte code like the java byte code [HD09]. Byte codes are executed by a virtual machine and contain more informations than binary programs. There are informations about data segments, types, method names, member names, entry points and exit statements [Vi03].

But all known decompilers are only able to generate code into one direction: From the binary program to the source code. They are working on the instructions of the binary program. Only a functional equivalence between the binary program and the decompiled source code can be achieved. In [PW93] such a decompiler is used to verify the equivalence between a binary program and the implemented source code of safety-related software. The decompiler translates the binary program and shows the functional equivalence by formal methods. But to compare the original source code and the decompiled source code they have to be translating both source codes into a formal representation.

One of the main problems is that many data are lost during compiling. But these data are required to restore the source code exactly. Therefore, this approach examines not only the decompiler phases, but also the compiler phases to gather the data that are normally lost.

Section 2 shows the symmetry between compilers and decompilers, while Section 3 gives an overview on this approach and describes the data that are required to restore the original source code. Section 4 concludes this paper.

2 Symmetry between a compiler and a decompiler

Modern compiler translates the source code in a binary program by a sequence of phases [Ah06]. The lexical analysis read the source code and split it into meaningful sequences called lexemes. For each lexeme it creates a token that contains the lexeme itself and an identifier of the token type. The sequence of tokens is handled by the syntax analysis. If it is possible to generate a syntax tree, the source code will be syntactical correct. The semantic analysis does verifications about types and language specific characteristics which cannot specify by a context free grammar. At the end of these frontend phases the source code is presented as an intermediate representation. All required symbolic informations are collected in the symbol table, which can access in every phase of the compiler. Optimizations will be performed on the intermediate representation and on the target code instructions before the binary program is generated.

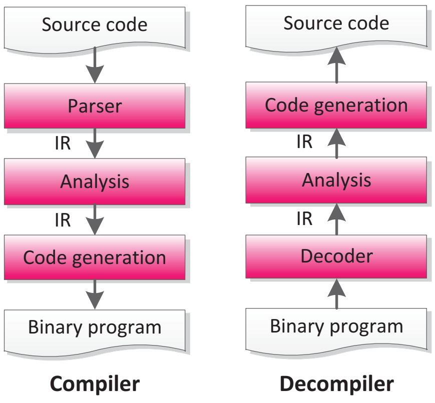

Figure 1: Symmetry between a compiler and a decompiler [Em07]

A decompiler generates the source code in a sequence of phases, too [Em07]. The decoder reads the binary program and splits the instructions from the data. It analyzes the control flow and the data flow and creates an intermediate representation of the binary program. Machine specific constructs will be replaced by corresponding constructs of higher programming languages and type informations are reconstructed. Programming language specific constructs will be recovered from the intermediate representation and the source code is generated.

Emmerik describes the symmetry between a compiler and a decompiler and consisders the similar techniques in the phases [Em07]. Figure 1 gives a short overview of the symmetry between a compiler and a decompiler. It shows that the decompiler is an inverse of a compiler.

3 Approach

In this approach the compiling and decompiling are considered together. Each phase of compilation has to be covered by a phase of decompilation. The symmetry between a compiler and a decompiler is the starting point of this approach. In contrast to a conventional compiler, the details of structure and format of the source code are not allowed to get lost during compile time. They have to be collected and stored in the binary program itself. Using this additional data, the original source code is reproducible.

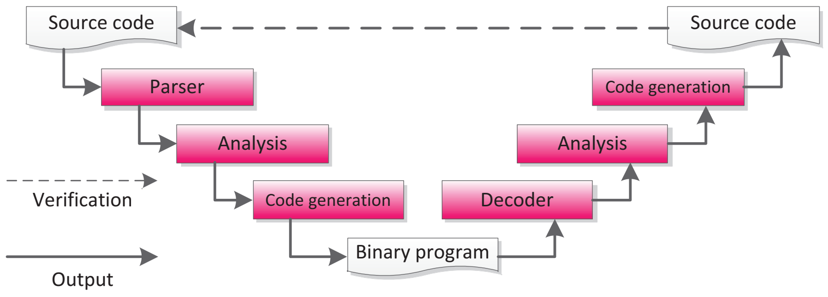

Figure 2: V-Model of the Two-Way-Compiler

3.1 Compiler phases

The compiler phases are used to transform a given source code to an executable binary program, which can be restored to the original source code back again. Figure 3 shows a simplified scheme of the compiler phases.

The lexical analysis reads the source code and generates a sequence of tokens. Normally, whitespaces and comments are ignored at this point of compilation. This approach uses an ignore list in addition to the symbol table. The ignore list collects all the ignored lexemes and their beginning positions. The syntax analysis works on a context free grammar and parses the token sequence into an abstract syntax tree. Special tokens that are lost during this phase are collected in the ignore list, too. The semantic analysis does type checking and fill the symbol table. It collects symbol informations like variable names and function names. Each entry holds the token informations and a data type at the end of this phase.

After the front end has collected the required data, the abstract syntax tree is transformed into a SSA form. This intermediate representation is used by many compilers [Cy89] and is qualified for decompilers, too [Em07]. The mapping between nodes in the abstract syntax tree and the SSA statements has to be reversible. Currently, there are not considered optimizations in this approach.

The back end generates the binary program out of the SSA form. In this approach the selection of target code instructions is very important, because the sequence of instructions has to identify the SSA form during the decompilation. Addresses for variables in the data sections are calculated and added to the corresponding symbol in the symbol table. The addresses of function entry points in the code sections are added to the symbol table, too. At least the binary is build and gets two additional sections. The first one contains the data from the ignore list and the second one holds the data from the symbol table.

The result of the compiler phases is an executable binary program that contains additional informations to restore the original source code back again. These informations are explicit collected data and informations contained in the structure of the binary program.

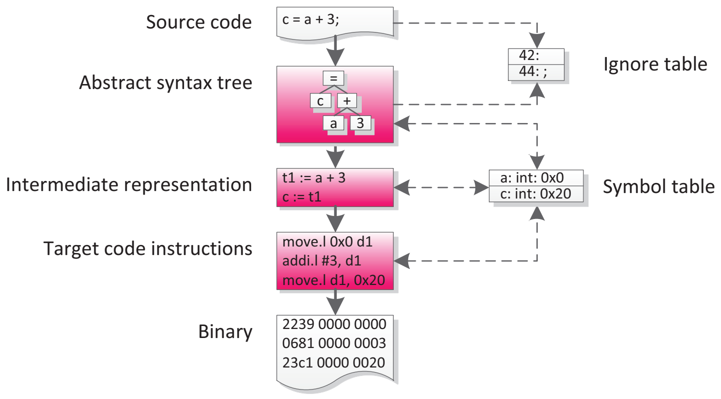

Figure 3: Simplified compiling scheme

3.2 Informations in binary programs

There are informations that can be gained from the binary program itself. These data can be evaluated by control flow and data flow analysis and this is also done by many decompilers [Vi03]. Control flow analysis [Al70] and data flow analysis [Ki73] are well known techniques and used for optimizations in a compiler, normally. Control structures, functions, function calls, operators and variables can be identified by using these techniques. The special selection of the target code instructions, the knowing of the programming language grammar and the knowledge of the mapping between the different representations make it possible to identify these informations and restore the corresponding lexemes in this approach.

To restore the exact source code, it is important that the compiler does not perform any optimizations in any phase. Optimizations would change the control flow and the data flow [SS08]. After optimizations the sequences of target instructions are modified. Thus, the decompiled control structures of the source code would not be the same as in the original source code. The reliability is more important than highly optimized programs in safety-related software. So, this restriction can be accepted for the compiler in this approach.

3.3 Lost data

Normally, many data are lost during the compiling process. In this approach these data are collected in every single phase of the compiling. The compiler saves whitespaces, formatting and comments of the source code in their ignore list during the lexical analysis. The names of functions and variables are lost, too. These data are collected in the symbol table of the compiler. In addition to the identifiers, the symbol table collects the memory addresses and the data types of the variables and functions. Typically, tokens like brackets are lost in the syntax analysis. These tokens are not in the abstract syntax tree and have to be stored in the ignore list, too. For example, this is necessary to restore arithmetical expressions that contain brackets exactly.

All collected data have to be included in two separate sections at the end of the binary program. The data from the symbol table and the informations of the ignore list are stored in these sections. Both sections are enclosed by special section markers to identify these sections during decompilation. To reduce storage space, the additional data sections are compressed and protected by a checksum.

3.4 Decompiler phases

The decompiler phases are used to restore the original source code out of the generated binary program exactly. Only binary programs can be handled that are generated from the compiler in this approach. Figure 4 shows a simplified scheme of the decompiler phases.

The decoding phase of the decompiler reads the binary program and separate instructions from the data. It identifies the symbol table section and the ignore table section. After decompressing these sections, the ignore list and the symbol table are filled with the existing data. The data from the restored symbol table and the restored ignore list are required in the following phases.

Because the decompiler has the same informations of the programming language grammar and the target code specification as the compiler, it knows the sequences of instructions and their parameters. Each identified sequence is transformed into the SSA form. The variable names and function names are obtained by the memory addresses and the data from the symbol table which were restored from the binary program. The symbol table contains the data types for variables, function parameters and function return values. It contains the names for variables and functions, too. After the transformation from the binary program to the intermediate representation, the control structures of higher programming languages are restored. The knowing of the programming language grammar and the knowledge of the mapping between the different representations makes it possible. A check for the syntax of the control structures are performed by transforming the SSA form into an abstract syntax tree. The syntax is correct, if the abstract syntax tree can be generated.

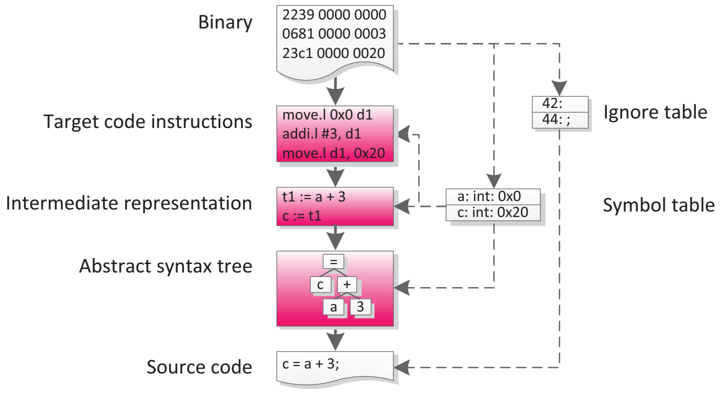

Figure 4: Simplified decompiling scheme

In the last phase the informations from the ignore list are used to restore the formatting and comments. The decompiler iterates the control structures and compares the positions in the ignore list. If the position is valid, it appends the ignored data to the decompiled source code. Inconsistent positions indicate an error during the decompilation. The decompiler uses the informations in the knowing programming language grammar to restore the identifiers for operators and control statements.

The result of the decompiler phases is the source code for the binary program that is the same as the original source code. It can be used to compare the decompiled source code with the original source code and shows the equivalence between the binary program and the source code.

4 Conclusion

This approach makes it possible to translate a source code into a binary program and back again. Through the simultaneous consideration of the compiler and the decompiler, as well as appending additional data to the binary program, it is possible to produce the original source code out of the binary program exactly.

One limitation of this approach is that no optimizations are done during compile time. This can be accepted, because the focus lies on safety-related software. A further disadvantage is the increased storage space of the generated binary program, which is created by the additional data.

However, the main advantage is that the verification between source code and binary program can be performed by a simple comparison between the original source code and the decompiled source code.

References

[Ah06] Aho, A. V.; Lam, M. S.; Sethi, R. and Ullman, J. D.: Compilers: Principles, Techniques, and Tools. Prentice Hall, 2nd edition, 2006.

[Al70] Allen, F. E.: Control flow analysis. In Proceedings of a symposium on Compiler optimization, 1970; pp. 1-19.

[Ci94] Cifuentes, C.: Reverse Compilation Techniques. PhD thesis, University of Queensland, 1994.

[Ch10] Chen, G.; Wang, Z.; Zhang, R.; Zhou, K.; Huang, S.; Ni, K.; Qi, Z.; Chen, K. and Guan, H.: A refined decompiler to generate c code with high readability. In 17th Working Conference on Reverse Engineering (WCRE), 2010; pp. 150-154.

[Cy89] Cytron, R.; Ferrante, J.; Rosen, B. K.; Wegman, M. N. and Zadeck, F. K.: An effcient method of computing static single assignment form. In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), 1989; pp. 25-35.

[Em07] Emmerik, M. V.: Static single assignment for decompilation. PhD thesis, University of Queensland, 2007.

[Ha62] Halstead, M. H.: Machine-independent computer programming. Spartan Books, 1962.

[HD09] Hamilton, J. and Danicic, S.: An Evaluation of Current Java Bytecode Decompilers. In 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2009; pp. 129-136.

[IEC10] IEC: Functional safety of electrical / electronic / programmable electronic safety-related systems - Part 3: Software requirements (IEC 61508-3:2010). International Electrotechnical Commission, 2010.

[Ki73] Kildall, G. A.: A unified approach to global program optimization. In Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL), 1973; pp. 194-206.

[PW93] Pavey, D. J. and Winsborrow L. A.: Demonstrating equivalence of source code and prom contents. The Computer Journal, 36(7), 1993; pp. 654-667.

[Sa66] Sassaman, W. A.: A computer program to translate machine language into FORTRAN. Proceedings of the Spring joint computer conference (AFIPS), 1966; pp. 235-239.

[SS08] Srikant, Y.N. and Shankar, P.: The compiler design handbook: optimizations and machine code generation. CRC Press, 2nd edition, 2008.

[Vi03] Vinciguerra, L.; Wills, L.; Kejriwal, N.; Martino, P.; and Vinciguerra, R.: An experimentation framework for evaluating disassembly and decompilation tools for C++ and java. In Proceedings of the 10th Working Conference on Reverse Engineering (WCRE), 2003; pp 14-23.