The 2nd International Conference on Computer Application and System Modeling (2012)

Analysis of Program Based on Function Block

Wu Weifeng
China National Digital Switching System Engineering & Technological Research Center
Zhengzhou, China
beewwf@sohu.com

***Abstract*-Basic block in program analysis plays an important role and it be applied to wide range, such as program compilation, program optimization, reverse engineering, program correctness verification, software security analysis and core module selection, etc. The binary executable analyzed by reverse engineering can contains indirect jump, however, the program analysis technologies based on basic block are not conducive to extract the target address information of indirect jump. In this paper, the presented analysis method based on function block is conducive to extract the target address of indirect jump and solve the problem of constructing the complete control flow graph. The correctness and validity of this method has been verified by the testing of spec2000 and spec2006 benchmark set, and it provides strong support for the research and application of reverse engineering.**

Keywords-Program analysis, Function block, Basic block, IA64

I. INTRODUCTION

In computer science, program analysis [1] is the process of automatically analysing the behavior of computer programs. The purpose of program analysis has three points: First, through the call relations between modules of program to grasp the running flow, so as to better understand the program. Second, system optimization for the purpose of the program through the tracking of key functions or run-time statistical information to find performance bottlenecks，thereby take further action to optimize the program. Finally, program analysis can be used for system testing and debugging, when the system tracks are more complex, and a BUG is hard to find, you can use some special data to construct a test case, and then compare the analyzed function call relationship with the actual function call relationship of run-time to identify the location of the error code.

Reverse engineering [2] is a software analysis process, which is concerned to translate legacy software to program which is represented in advanced expression form, in order to better understand the software. Legacy software is usually binary executable, when the binary to be translated contain indirect jump instructions, which must be dealt with, will hinder the construction of the program. Because that purely static analysis is very difficult to completely gain the target addresses of the indirect jump instruction, and dynamic analysis can only get one of the target addresses of indirect jump instruction each time because of the limitation of test data. In this paper, we present a program analysis method based on function block analysis, which is based on program debugging to extract all the target address information of indirect jump instruction, in order to build complete control flow graph.

This paper is organized as follows: Section 2 describes the art of reverse engineering. Section 3 describes the situation that basic block can not satisfy the analysis application. Section 4 presents the function block analysis method, gives the definition of function block and function block partitioning algorithm, and gives analysis of indirect jump generated by switch and non-switch structure. Section 5 gives the experimental results based on function block, and Section 6 presents some conclusions.

II. RESEARCH SITUATION

The key of reverse engineering analysis is constructing control flow graph of binary executable program, however the construction of program's control flow graph depends on disassembly.

Disassembly algorithm can be divided into linear scan algorithm and recursive scan algorithm based on control flow. Linear scan algorithm faces the issue of distinction between code and data. The main problem of recursive scanning algorithm is how to obtain the subsequent control flow of indirect jump or indirect call to continue decoding.

To address these issues, scholars have conducted extensive research. Reference [3] presents a method based on instruction credibility and the credibility of the instruction sequence guiding linear scanning, although some illegal instructions can be excluded from the interference of disassembly, essentially, it does not solve the problem of distinction between code and data, the data can still be identified as instruction, and does not give description of how to establish correct relationship between indirect jump or indirect call and code identified from the remainder binary code. Cifuentes and Emmerik has been proposed a static method [4] to identify indirect jump table based on slicing and expression replacement, but this solution depends on the compiler version, and the indirect jumps of program do not all dependent on the jump table (such as the destination address of goto statement only be obtained when running, pointer of function call involving pointer arithmetic, various indirect jump embedded by hand at assembly-level); Reference [5] describes two passes disassembly method which using linear and recursive scan disassembly mutual authentication, although it is able to report errors in the disassembly results and to limit the

© the authors

spread of the error, does not guarantee the integrity of the program decoding.

Data flow analysis statically calculates information about the program variables from a given program representation, it can assist to solve the problem induced by indirect jump. Data flow analysis requires a precise control flow graph to work on. However, we can't have a precise control flow graph before resolve the subsequence problem of indirect jump. This seemingly paradoxical situation has been referred to as an inherent "chicken and egg" problem in the literature [6,7]. Earlier works [6,8,9,10,11] have shown that data flow analysis can be used to augment the results of disassembly, but no conclusive answer was given on the best way to handle states with unresolved control flow successors during data flow analysis.

III. ANALYSIS BASED ON BASIC BLOCK

A basic block is a linear sequence of program instructions which has one entry point and one exit point, the last instruction must be executed when the first instruction is executed [12]. Describe the partitioning algorithm of basic block following.

Algorithm 1: The three-address instructions sequence is divided into basic blocks.
INPUT: A three-address instructions sequence.
OUTPUT: A set of basic block corresponding to the input sequence, and each instruction can only belongs to a basic block.

1. First determine a set for the head instruction, which is the first instruction of basic block, using the follow rules.
   1) The first instruction of input sequence is the head instruction;
   2) The target instruction of any conditions or unconditional branch instructions is the head instruction;
   3) The instruction which immediately follows the branch instruction is the head instruction.
2. The basic block corresponding to each head instruction is an instructions sequence, which starts at its head instruction and ends at the next head instruction (not included) or the end instruction of the input sequence.

The following analysis are carried out for the program switch.c, which accomplish the function of statistics recorded scores rank number, it contains switch branch structure, as follows:

```
main()
{
    int grade;
    int aCount = 0, bCount = 0, cCount = 0, dCount = 0, fCount = 0;
    printf("Enter the letter grades.\n");
    printf("Enter the EOF character to end input.\n");
    while ( ( grade = getchar() ) != EOF) {
        switch (grade) {      /* switch nested in while */
            case 'A': case 'a':   /* grade was uppercase A */
                ++aCount;             /* or lowercase a */
                break;
            case 'B': case 'b':   /* grade was uppercase B */
                ++bCount;             /* or lowercase b */
                break;
            case 'C': case 'c':   /* grade was uppercase C */
                ++cCount;             /* or lowercase c */
                break;
            case 'D': case 'd':   /* grade was uppercase D */
                ++dCount;             /* or lowercase d */
                break;
            case 'F': case 'f':   /* grade was uppercase F */
                ++fCount;             /* or lowercase f */
                break;
            case '\n': break;
            default:              /* catch all other characters */
                printf("Incorrect letter grade entered.  Enter a new grade.\n ");   break;
        }
    }
    printf("\nTotals for each letter grade are:\n");
    printf(" A: %d\n B: %d\n C: %d\n D: %d\n F: %d\n ", aCount, bCount, cCount, dCount, fCount);
    return 0;
}
```

A. *Competent Situation*

Using compiler gcc, whose version is 3.2.3, with –O0 option compiles switch.c on IA64 computer to generate binary executable a.out, the assembly codes generated by the recursive scanning disassembler corresponding to a.out are shown in Figure 1, in order to save space replace memory address of instruction with the serial number.

According to the Algorithm 1 the division of the generated assembly codes is shown in Figure 1, divided into 13 basic blocks. Since the instruction L054 in the basic block B8 is indirect jump instruction, we don't know the successors of the basic block B8, therefore, the control flow graph constructed for a.out is incomplete.

To build complete control flow graph, the successors of basic block B8 need to be determined, according to the second rule of Algorithm 1 we should obtain target addresses of instruction L054, namely the value of register b6.

The general way to get the value of register B6 at instruction L054, is using Mark Weiser's program slicing [13] technology to obtain all the instructions related to the state of b6 at instruction L054, namely the slice (L054, b6) shown in Figure 2.

From the slice (L054, b6) we can know that the value of register b6 at L054 depends on the value of register r35 at L045 and the value stored in memory cells of r35+8 and the value of register r0 at L047 and the value of register r1 at L048. However, register r35 stores the value of register

© the authors

r12, and r35 only be defined at L002. By the introduction of Itanium architecture [14] know that: register r0 is the constant register, whose value is always 0; register r1 is a special register, which remains unchanged in the program, stores the global data pointer; register r12 is also a special register, which remains unchanged in the current procedure, stores the stack pointer. Thus, only the instructions in the basic block B8 can decide the value of register b6 at L054.

B. Incompetent Situation

Using the same compiler with –O2 option compiles switch.c to generate binary executable b.out, which different from a.out. The assembly codes generated by the recursive scanning disassembler corresponding to b.out are shown in Figure 3.

According to the Algorithm 1 the division of the generated assembly codes is shown in Figure 3, divided into 11 basic blocks. Since the instruction L035 in the basic block B6 is indirect jump instruction, we don't know the successors of basic block B6, therefore, the control flow graph constructed for b.out is incomplete.

Similarly, to build complete control flow graph, the successors of basic block B6 need to be determined, namely to get the value of register b6. The slice (L035, b6) shown in Figure 4.

From the slice (L035, b6) we know that the value of register b6 at L035 depends on the value of register r1 at L018 and the value of register r8 at L021. Because register r8 is used to store return value of function [14], so the assignment to r8 is implicit manifest. As above analysis the value of register r1 is a global data pointer, which remains unchanged in the program. There are two important instructions in slice (L035, b6): one is L024, the basic blocks B5 and B6 will not be executed if the predicate register p7 is true, however, the value of predicate register p7 is controlled by register r8; another is L029, if the predicate register p6 is true then the basic block B6 will not be executed, the value of predicate register p6 is also controlled by the register r8. Therefore, in essence, the value of register r8 at L021 controls the value of register b6 of L035 instruction. Instructions of basic block B6 can not determine the value of register b6 at L035, although the union of basic blocks B3, B4, B5 and B6 can decide the value of register b6, it contains a number of irrelevant and redundant instructions.

IV. FUNCTION BLOCK

Programs running on your computer, no matter large (the whole program or module) or small (function or program fragment) must contain three parts: input data, executable code and output data. If without obvious input and output data, the memory state before and after code execution can still be considered as the input data and output data. A procedure segment with clear function is composed of input data, executable code and output data.

A. Definition and Partitioning Algorithm of Function Block

The analysis unit frequently used in the field of program analysis is basic block, but the division of the basic block is mechanical and rigid, it basically has no meaning to the code's function because that the basic block can not give a description of the function with a clear meaning.

Function block is a minimum sequence of statements related to interested variable at specific point of procedure, which can contain one or more basic blocks or by one or more basic blocks and parts of one to a few basic blocks, with one entry point and multiple exit points. Function block is a concept of program division proposed based on the description of code's function for more favorable analysis of the code.

The control flow graph used to represent the program divided by function blocks is defined as: the nodes represent function blocks and the edges represent control flow paths. When function block contains no transfer instructions uses node 5-(a) to represent it. If function block contains transfer instructions uses node 5-(b) to represent it , and the sequence of edges issued from the node 1, 2, ..., n are control flows corresponding to the transfer instructions of the function block, and the order of edges corresponding to the sequence of transfer instructions of the function block, such as the transfer instruction corresponding to edge 1 is prior to the transfer instruction corresponding to edge 2, the edge just below the node corresponding to control flow at the end of the function block. The classification of nodes is shown in Figure 5.

Before describe the partitioning algorithm of function block, we need to know four concepts: 1) interest point - the instruction surrounded to construct function block, the interest points of slicing shown in Figure 2 and Figure 4, respectively, are instruction L054 and instruction L035; 2) function block control instruction - the instruction which controls the execution of function block , it is associated with input data of the function block, compare instruction L043 is the function block control instruction of interest point L054; concern interest points L035, the function block control instruction is compare instruction L028; 3) control variable - the variable which plays a key role in function block control instruction, the control variable of the function block control instruction L028 is r8; 4) function block input variable - the variable which provides the input data for function block.

Algorithm 2: Build function block of a given interest point.
INPUT: A program and the interest point.
OUTPUT: Function block of the interest point, function block input variable and the range of input data.

1. First of all, according to interest point to determine function block control instruction.

2. Starting from the interest point to slice for interested variables, until it encounters simple setting instructions (eg: mov r35 = r12, mov r1 = r32, mov r32 = r1) which related

© the authors

to special registers (such as: r0, r1, r12, etc).

3. Statistics on the slice obtained in Step 2 about the simple setting instruction related to special registers, and remove them from the slice.

4. If the slice after step 3 does not contain function block control instruction, then turn to step 7.

5. For the remaining instructions of the slice after step 3, which don't belong to the sequence starts at the function block control instruction and ends at the interest point, delete that only associated with control variable.

6. Extracting the range of control variable according to the function block control instruction, and the control variable is defined as the function block input variable.

7. From function block control instruction to check forward and backward for whether there exist equivalent instruction sequences (such as: instruction sequence L041 ~ L042 and L045 ~ L046 in Figure 1) related to the data stored with the control variable (also interpreted as the memory cell). If there exist equivalent instruction sequences extract the range of control variable according to the function block control instruction, then assign the range of control variable to the variable Vend (Eg: assign the range of control variable r16 of instruction L043 to the variable r14 of instruction L046), which be defined at the end of the equivalent instruction sequence in the slice, and delete the equivalent instruction sequence from the slice, finally define variable Vend as the function block input variable. Otherwise, define function block input variables as NULL.

8. The remaining instructions of the slice constitute the function block, exit.

B. Case Study

The instructions sequence L047 ~ L054, which is a part of the basic block B8 shown in Figure 1, constitute a function block to complete the calculation function of register b6 at the instruction L054, see Figure 6-(a). From Figure 4 we know that the value of register b6 at the instruction L035 is essentially controlled by the value of register r8, which last appeared at L028 before instruction L035, moreover instructions L028 and L029 together form a conditional jump module, in addition, the instructions sequence L030 ~ L035 data dependent on instructions L026 and L027. Therefore, basic block B6 and part of the basic block B5 in Figure 3, namely the instructions sequence L026 ~ L035 constitute a function block to complete the calculation function of register b6 at the instruction L035, see Figure 6-(b). The function blocks obtained by the above analysis are the same with that built by Algorithm 2.

The input variable of the function block 6-(a) is register r14 and the input data range is [0, 92]. The input variable of the function block 6-(b) is register r8 and the input data range is [0, 92]. In terms of output data of function blocks 6-(a) and 6-(b) is the value of register b6.

If properly set the value of input variable at the beginning of function block, namely before the first instruction execution, then execute the function block and repeat the above operations until complete traversing the input data, you can obtain the complete output data. The output data of function blocks 6-(a) and 6-(b) are shown in Figure 7-(a) and Figure 7-(b) respectively.

After obtain the target address set of indirect jump through debugging based on function block, which is shown in Figure 7, we can re-disassemble binary executable b.out and a.out to get the assembly codes corresponding to Figure 8 and Figure 10.

The complete control flow graph corresponding to Figure 8 and Figure 9 are Figure 11-(a) and Figure 11-(b). They cover all of the program's instruction code respectively.

A program fragment contains an indirect jump, which does not produced by switch structure, shown in Figure 9. For interest point Ln+11 using the Algorithm 2 to process this fragment can obtain its function block, which is composed of the instructions sequence Ln+8 ~ Ln+11, and its function block input variable is NULL. Analysis program fragment shows that the target address of this indirect jump controlled by the instructions sequence Ln+4 ~ Ln+7, according to the execution path, which can reach this function block, debugging the program can obtain the target address.

V. EXPERIMENTAL RESULTS

Using gcc compiler, whose version is 3.2.3, compile the program switch.c with option –O0, –O2, –O1, –O3 and –O4 to generate binary executable a.out, b.out, c.out, d.out and e.out. In addition to a.out and b.out, using the function block analysis method described in section 4 to analyze c.out, d.out and e.out, the complete control flow graph of each binary executable can be constructed.

Using gcc compiler, whose version is 3.2.3, compile the 15 C programs of spec2000 benchmark set (gzip, vpr, gcc, mcf, crafty, parser, gap, perlbmk, vortex, bzip2, twolf, mesa, art, equake, ammp) and 14 C programs of spec2006 benchmark set (perlbench, bzip2, gcc, mcf, gobmk, hmmer, sjeng, libquantum, h264ref, milc, lbm, sphinx3, 998 . specrand, 999.specrand) with option –O0~4 to generate 145 binary executables. Using g95 compiler, whose version is 0.91, compile the 10 Fortran programs of spec2000 benchmark set (wupwise, swim, mgrid , Applu, galgel, facerec, lucas, fma3d, sixtrack, apsi) and 10 Fortran programs of spec2006 benchmark set (bwaves, gamess, zeusmp, gromacs, cactusADM, leslie3d, calculix, GemsFDTD, tonto, wrf) with option –O0~3 to generate 80 binary executables. The total number of generated test case is 225.

Applying function block analysis method to process the 225 test cases of Spec2000 and Spec2006 benchmark set, all of the function blocks corresponding to indirect jump can be successfully divided, and it provides strong support for constructing the complete control flow graph of test cases.

© the authors

VI. CONCLUSIONS AND FUTURE WORK

Basic block in program analysis especially reverse engineering plays an important role, however, the program analysis technologies based on basic block are not conducive to extract the target address information of indirect jump. In this paper, the presented analysis method based on function block is conducive to extract the target address of indirect jump and solve the problem of constructing the complete control flow graph. The correctness and validity of this method has been verified by the testing of spec2000 and spec2006 benchmark set.

This paper presents the partition algorithm of function block based on introduces the inadequate of analysis method based upon basic block, but the detail about how to use available function block, function block input variable and the range of input data effectively extract the desired program Information do not discuss. This is the urgent work to solve.

REFERENCES

[1] "Program analysis". http://en.wikipedia.org/wiki/Program_analysis. Retrieved 2011-03-14.

[2] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, 7(1):13–17, January 1990.

[3] Wu Jinbo. Research on a static disassembly algorithm based upon control flow. Computer Engineering and Applications.2005,30,pp.89-91.(China)

[4] Cifuentes Cristina, Emmerik Mike Van. Recovery of jump table case statements from binary code[c]／Proc of the 7th Int Workshop on Program Comprehension. Washington, IEEE Computer Society, 1999.

[5] Zeng Ming. A relocation information-based revisited method for disassembly. Computer Science. 2007, 34(7), pp.284-287. (China)

[6] Theiling, H. Extracting safe and precise control flow from binaries. In 7th International Workshop on Real-Time Computing and Applications Symp (RTCSA 2000), pp. 23−30. IEEE Computer Society, Los Alamitos, 2000.

[7] Schwarz, B., Debray, S.K., Andrews, G.R. Disassembly of executable code revisited. In 9th Working Conf. Reverse Engineering (WCRE 2002), pp. 45−54. IEEE Computer Society, Los Alamitos, 2002.

[8] Balakrishnan, G., Reps, T.W. Analyzing memory accesses in x86 executables. In Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5−23. Springer, Heidelberg, 2004.

[9] Kastner, D., Wilhelm, S. Generic control flow reconstruction from assembly code. In 2002 Jt. Conf. Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES 2002-SCOPES 2002), pp. 46−55. ACM Press, New York, 2002.

[10] Kinder, J., Veith, H. Jakstab: A static analysis platform for binaries. In Gupta, A., Malik, S.(eds.) CAV 2008. LNCS, vol. 5123, pp. 423−427. Springer, Heidelberg, 2008.

[11] De Sutter, B., De Bus, B., De Bosschere, K. Link-time binary rewriting techniques for program compaction. ACM Trans. Program. Lang. Syst. 27(5), 882−945, 2005.

[12] Frances E. Allen. Control Flow Analysis. In ACM SIGPLAN Notices - Proceedings of a symposium on Compiler optimization, Volume 5 Issue 7, July 1970.

[13] Mark Weiser. "Program slicing". Proceedings of the 5th International Conference on Software Engineering, pages 439－449, IEEE Computer Society Press, March 1981.

[14] James S. Evans, Gregory L. Trimper. "Itanium Architecture for Programmers: Understanding 64-Bit Processors and EPIC Principles". Prentice Hall, PTR. Appendix D

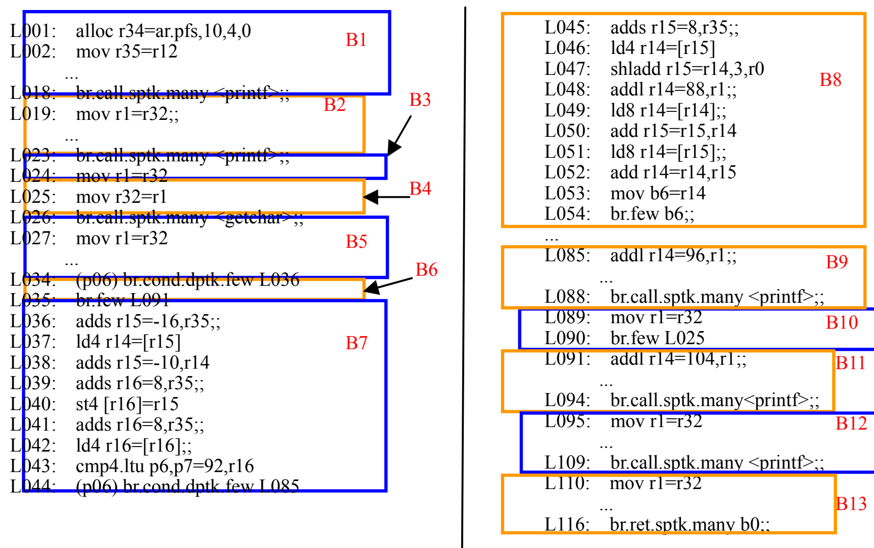

Figure 1. Assembly codes of a.out generated by disassembler and partition based on basic block

© the authors

```
L002:   mov r35=r12
        ...
L045:   adds r15=8,r35;;
L046:   ld4 r14=[r15]
L047:   shladd r15=r14,3,r0
L048:   addl r14=88,r1;;
L049:   ld8 r14=[r14];;
L050:   add r15=r15,r14
L051:   ld8 r14=[r15];;
L052:   add r14=r14,r15
L053:   mov b6=r14
L054:   br.few b6;;
```

Figure 2. Instructions of slice (L054, b6)

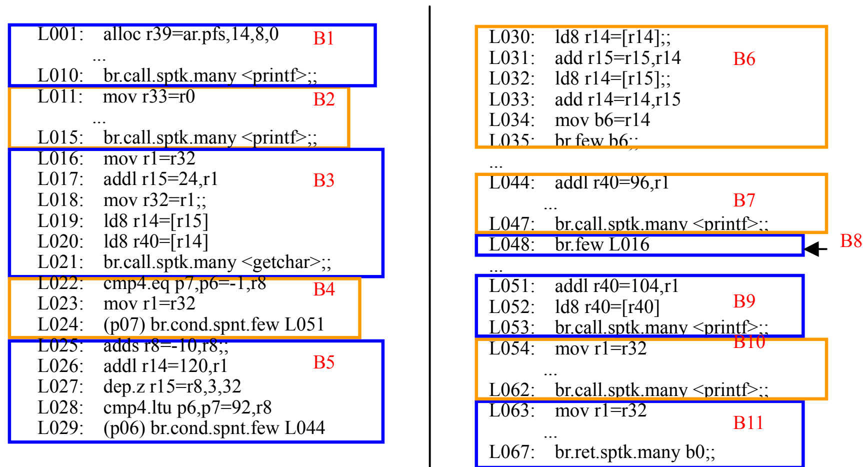

Figure 3. Assembly codes of b.out generated by disassembler and partition based on basic block

```
L018:   mov r32=r1;;
L021:   r8=br.call.sptk.many <getchar>;;
L022:   cmp4.eq p7,p6=-1,r8
L023:   mov r1=r32
L024:   (p07) br.cond.spnt.few L051
L025:   adds r8=-10,r8;;
L026:   addl r14=120,r1
L027:   dep.z r15=r8,3,32
L028:   cmp4.ltu p6,p7=92,r8
L029:   (p06) br.cond.spnt.few L044
L030:   ld8 r14=[r14];;
L031:   add r15=r15,r14
L032:   ld8 r14=[r15];;
L033:   add r14=r14,r15
L034:   mov b6=r14
L035:   br.few b6;;
```

Figure 4. Instructions of slice (L035, b6)

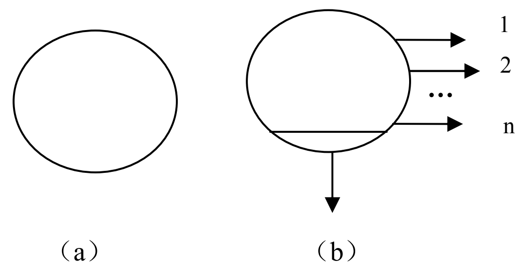

Figure 5. Representation of nodes about function block

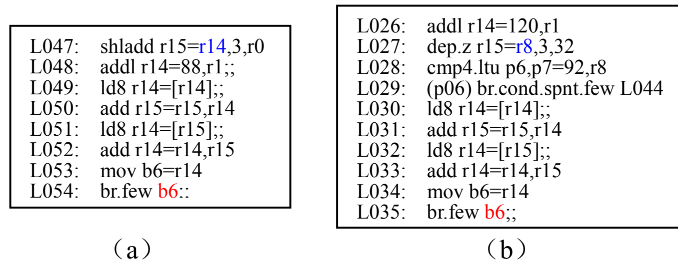

Figure 6. Function block

© the authors

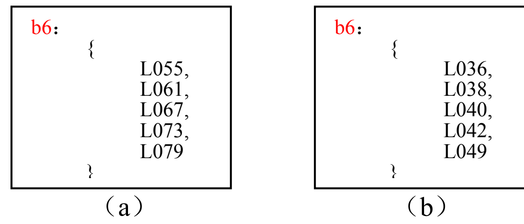

Figure 7. Output data of function block

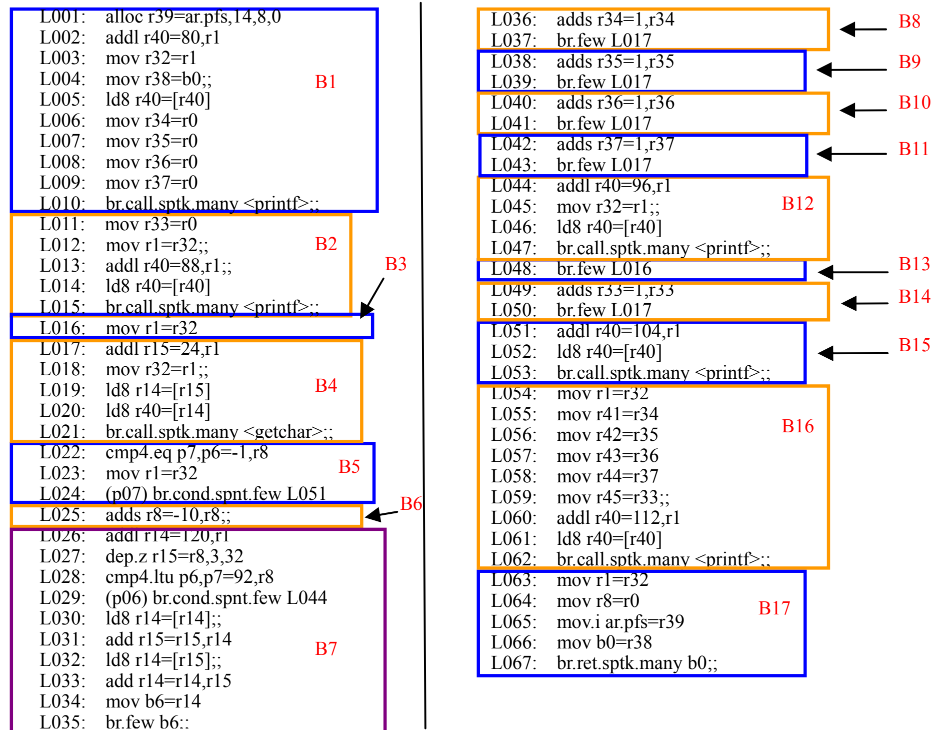

Figure 8. Complete assembly codes of b.out and partition based on function block

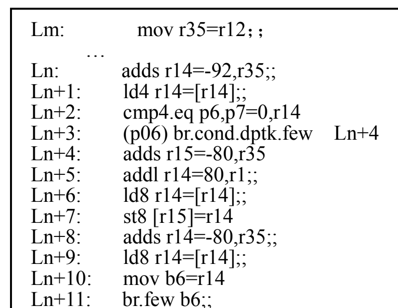

Figure 9. A program fragment contains non-switch indirect jump

Published by Atlantis Press, Paris, France.
© the authors

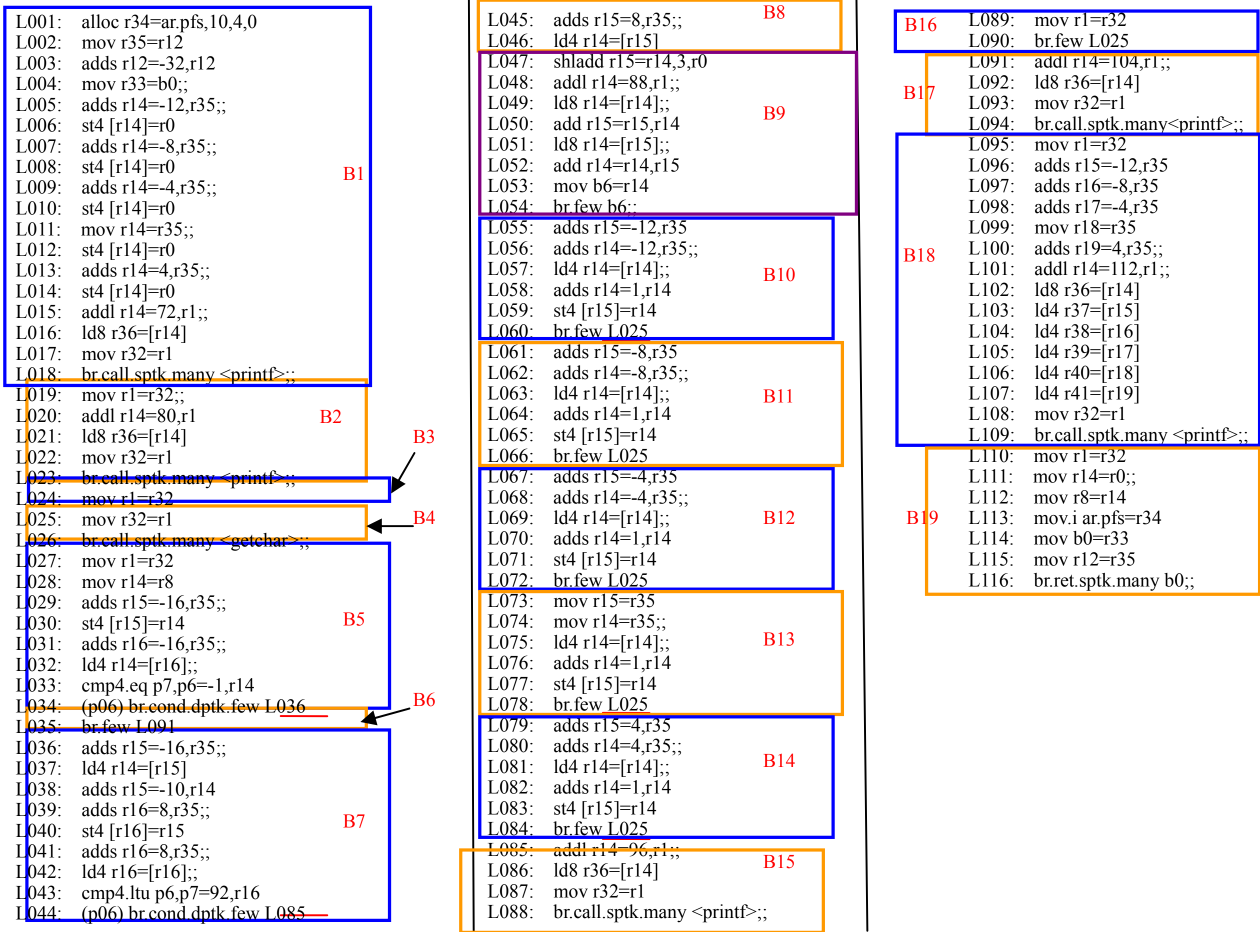

Figure 10. Complete assembly codes of a.out and partition based on function block

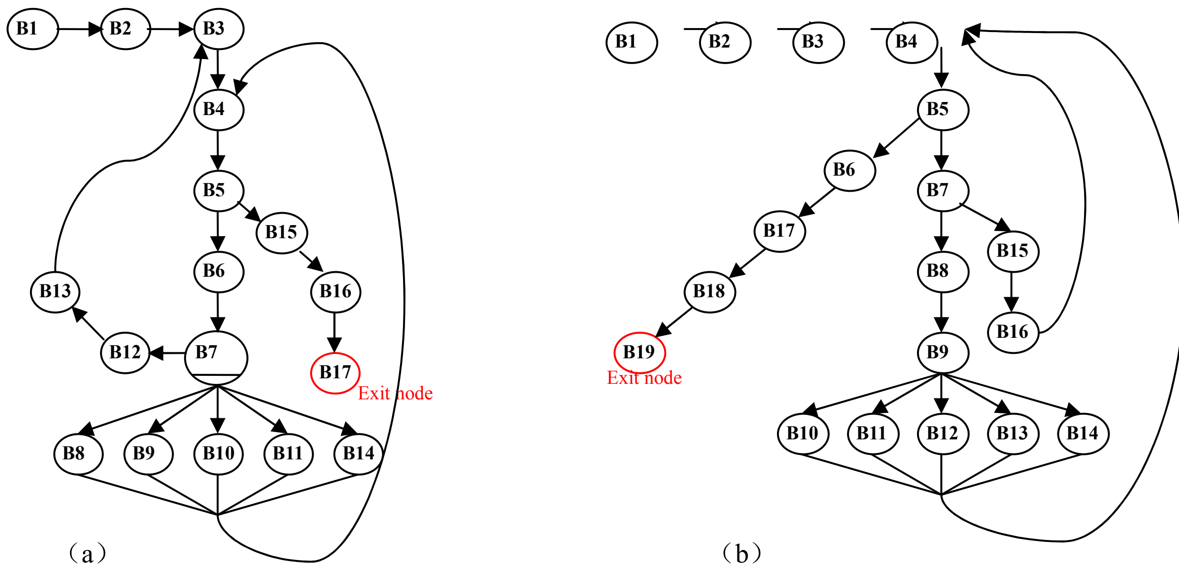

Figure 11. Complete control flow graph

Published by Atlantis Press, Paris, France.
© the authors