Binary analysis and dynamic jumps

Gina-Adriana DOBRESCU and Maxime MEIGNAN

École Nationale Supérieure d’Informatique et de Mathématiques Appliquées de Grenoble, France
Email: gina-adriana.dobrescu@ensimag.grenoble-inp.fr
maxime.meignan@ensimag.grenoble-inp.fr

Abstract - Binary analysis is a very complex process that helps us to understand, anticipate and solve different software problems. This paper describes a study for the recognition of switch statement via binary analysis. Generally, compilers choose different data structures in order to obtain faster results. For experimentations we used GCC and Clang compilers with IDA software, a multiprocessor disassembler and debugger. Using the IDAPython API, we created a plugin for the IDA software, that aims to improve the routine of switch statement recognition.

Keywords: binary analysis; research; switch statement; dynamic jumps; compilation; assembly; reverse engineering; IDA; GCC; Clang

1. INTRODUCTION

Dealing with binary machine code represents a big challenge for many reasons: from being able to understand, to make observations and even trying to improve some functions. Analysing binary code allows us to understand how a program works, without knowing its source code. This "reverse engineering" process contains two important phases. First, the disassembling process, which translates binary code into assembly language. Second, the construction of the Control Flow Graph (CFG), which is a visual representation of every possible sequence of a program execution.

Nevertheless, a program might contain dynamic jumps, in other words the code executed after a jump instruction depends on a value which is only known at runtime. Those dynamic jumps represent a real problem for the construction of CFGs, because the common CFG construction algorithm use branching instructions to create the nodes of the graph.

Here, we are interested in dynamic jumps generated by compilers in order to translate a switch statement. Switch statements sometimes use jump tables (also known as branch tables) as a method of transferring program control flow (branching) and to link each branch label to a memory location.

How do some compilers like GCC and Clang make their choices and how do they use these data structures to represent switch statements in an assembly language? After compiling many types of switch statement with different options, we tested the results with binary analysis softwares like Hex-Rays’ IDA Pro and Carnegie Mellon University’s Binary Analysis Platform (BAP)\(^2\).

The goal of our research is to help the construction of the CFG, by offering some effective solutions for the construction of the switch statement recognition algorithm.

2. COMPILERS’ BEHAVIOR

We wrote many different examples in C language using the switch statement(see annex, section switch example). GCC(version 4.8.2)\(^3\) and Clang(version 3.4)\(^4\) compilers provide a variety of optimization options that we used for our experimentations:

- \(-O0\) : optimization for compilation time (default)
- \(-O3\) : optimization for execution time
- \(-Os\) : optimization for code size

The code produced by each compiler was disassembled with objdump\(^5\). The sections below present our observations on the most relevant examples.

2.1. Switch statement with consecutive cases

First, we did the compilation on a simple switch with consecutive case labels, as we can see in FIGURE 1.

For this sample code, the assembly output of each compiler using the \(-O3\) optimization is presented in Figure 2.

\(^{1}\)https://www.hex-rays.com/products/ida/
\(^{2}\)http://bap.ece.cmu.edu
\(^{3}\)https://gcc.gnu.org
\(^{4}\)http://clang.llvm.org
\(^{5}\)http://www.gnu.org/software/binutils/
int main ( int argc , char* argv[] ) {
    switch ( atoi ( argv[1] ) ) {
        case 0:
            printf( " zero "); break ;
        case 1:
            printf( " un "); break ;
        case 2:
            printf( " deux "); break ;
        case 3:
            printf( " trois "); break ;
        case 4:
            printf( " quatre "); break ;
        case 5:
            printf( " cinq "); break ;
        default :
            printf( " autre "); break ;
    }
    return 0;
}

FIGURE 1: switch statement with consecutive cases

We can see that GCC and Clang generate a similar code.

- First they compare the control value of the switch statement (ie eax) with the immediate value 5.
- Then, if this value is above 5, a branching instruction (ie ja) redirect the program execution to the code corresponding to default.
- Else, the program uses an indirect jump (instructions 8048472 and 8048490 respectively), accessing a jump-table indexed by eax.

A jump-table is an array stored in the program memory, that contains the addresses of the code corresponding to each case label. The program can execute this code by accessing this jump-table with the correct index. The main difference between GCC and Clang is that the latter, without using any particular optimization option, stores in the execution stack the value of any register used by the program before each jump, and restores the necessary ones after the jump.

2.2. Switch with sparse cases

A switch statement with non-consecutive cases, containing the cases 0, 1, 4, 12, 24 and 2048 for example, is compiled very differently. As we can see in the Figure 3, both compilers do not use a jump-table, but a sequence of compare and jump instructions to find the control value of the switch, and execute the corresponding code.

After some tests, we conclude that compiling with any compiler or options will give the same pattern.

Also, this solution does not requires any dynamic jump, and therefore the CFG construction will not encounter problems.

Note : The assembly code is equivalent to the C code Figure 4.

FIGURE 2: Assembly code produced by each compiler

FIGURE 3: Assembly code for sparse cases
Binary analysis and dynamic jumps

2.3. Other patterns

Compilers may also provide an "hybrid solution" in order to compile some switch statements. For instance, a switch with cases 1 to 10 and 100 to 200 will be translated into two dynamic jumps, each one using a jump-table. For another example, a switch with a combination of sparse cases and a consecutive range of cases will result in the use of a sequence of compare/jump instruction for GCC, and a mix between the two solutions for Clang.

2.4. A word about performances

The use of a jump-table has a substantial cost, as it requires an additional memory access, compared with using a sequence of compare/jump instructions. But it needs less instructions to correctly redirect the program execution. Therefore, quality compilers like GCC and Clang are aware about the cost of each of these solutions, and choose between the two according to the number of cases they encounter. The minimum number of consecutive cases required by a compiler in order to adopt the jump-table solution depends on the machine, but is generally around 4 for both compilers.

Yet, GCC offers us to modify this threshold with the option --param case-values-threshold=N

3. IDA AND BAP ANALYSIS

Once studied most of the types of the switch statement compilation, we are going to focus on the efficiency of the binary analysers, by highlighting the behavior face to each type of binary. Then, our goal is to discover which of these programs pose problems to the prediction of their execution flow.

3.1. IDA

IDA Pro is a disassembler software which uses advanced algorithms and heuristics to disassemble, build the CFG of a program and even provides algorithms of decompilation. When we use CLANG with O3 compiling option, the CFG reconstruction is simpler than without options, as the compiler won’t use intermediate variables. A simplified example of the CFG after the compilation with CLANG and the option -O3 on the "six consecutive cases example" is represented in Figure 5. As we can see in this figure, the structure of the CFG is easy to understand. The compiler tests if the control variable of the switch statement is in the [0,5] range. In this case, the execution flow is redirected to the code of the jump table, #CodeJumpTable. Otherwise, the default code is executed(#CodeDefault).

We will obtain the same CFG for all the optimization options of GCC compilation.

Everytime a program contains a jump table and IDA is able to recognize its complete CFG, the blocks of the CFG contain comments related to the switch statement(see figures 6 and 7). For example, a switch statement with consecutive cases compiled with GCC(using any optimization) or with CLANG(using any optimization different of -O0), IDA comments the jmp instruction as a switch statement, and the nodes as cases:

FIGURE 4: Equivalent code in C

FIGURE 5: CFG after the Clang compilation with O3 option

FIGURE 6: Clang - Os optimization

FIGURE 7: IDA - Os optimization

uint32_t eax = atoi(argv[1]);
if (eax == 4)
go to code_4;
if (eax < 4){
    if (eax == 0)
go to code_0;
    if (eax != 1)
go to code_default;
    else
        go to code_1;
}
if (eax == 24)
go to code_24;
if (eax == 2048)
go to code_2048;
if (eax == 12)
go to code_12;
else
go to default;

FIGURE 5: CFG after the Clang compilation with O3 option

- ENSIMAG 2ND YEAR INTENSIVE PROJECT -
On the other side, even though it is able to disassemble a whole program, IDA does not reconstruct the CFG when there are too many intermediate variables for the control value of the switch statement. More, when the switch statement is implemented as a sequence of compare/jump instructions, IDA doesn’t write any related comment. In the Figure 8 we can see that IDA is only able to disassemble all the blocks, but there is no edge between the jump table block and the cases of the switch statement. In this circumstance, eax is saved before the branching and restored after. Then it is used for the indexation of the jump table.

3.2. BAP

Binary Analysis Platform, also known as BAP, is an open-source software suite specialized in binary analysis, which provide an intermediate language (BIL) to explicitly represent program instructions. It can be used for construction of CFG, or program verification. Regarding to the CFG, BAP can construct a program’s one whenever IDA Pro is able to. For the examples where IDA recover all the nodes of the graph but it is unable to predict a dynamic jump, BAP fails to construct a CFG, even partially.

4. SWITCH STATEMENT RECOGNITION

As we have seen in the examples above, binary analysers like IDA and BAP come across problems to reconstruct the CFG when the control variable of a switch statement goes through various registers and memory zones before the indirect jump. Without knowing the range of possible values for this variable, they are not able to predict the destinations of the possible jumps. The aim of our work is to help the CFG construction of IDA, by providing a different solution than a complex value analysis, and writing a plugin which would identify the use of jump-tables in compiled code.

4.1. The plugin

First, we need to determine what are the criteria of a compiled switch statement using a jump-table, independently of which compiler or optimization options are used. We wrote various code samples in C, and compiled each of them with different combinations of compiler and options, in order to analyse them. This allowed us to conclude on the following statements:

1. The jump-table is stored in the .rodata (Read-Only) segment of the program.
2. It contains addresses which belong to the .text (Code) segment
3. All addresses of a same jump-table belong to the code of a same function
4. The number of entries in the jump-table is greater or equal than the number of cases it handles. (some entries may point to the default code)
5. The code using the jump-table matches one of the following patterns:
   - mov reg, @JumpTable[reg*4]
   - jmp @JumpTable[reg*4]
6. By tracing the CFG back, we always find at least one couple of compare and jump instructions that constrains the possible values of the jump-table index.

Representing the result of the observations on the compilation, these criteria will lead to the development of a heuristic which try to recognize the use of a jump table and recover the missing edges of the CFG.

For the implementation of the plugin, we used IDAPython, an extension of IDA, that allows us to write Python scripts to interact with the CFG and automate every actions.

The first step of our algorithm is to detect potential usage of a jump-table by searching instructions that match the pattern 5. Then, we read the potential jump-table addresses in those instructions, and we verify the criteria 1, 2 et 3 on this address, while memorizing all the jump table data. Finally, we write the right comments in the blocks corresponding to the cases

---

6 A segment is a portion of address space in a program
7 ‘4’ is the size of an address in bytes for x86
and the dynamic jump, and we color those blocks with the same color in order to represent the edges between them.

Here is an example of the output of the plugin.

![Plugin applied on a switch with two JumpTables](image)

### 5. EXPERIMENTATIONS

To test the efficiency of our script, we verified if it can identify the use of a jumptable whenever the tested program contains one, but also that our plugin does not raise any false-positive match. We wrote our own benchmark, organizing the tests in three categories: first, the binaries containing a switch statement recognized by IDA’s analysis, the ones IDA does not recognize, and binaries that contain structures that are close to jumptables and switch statements in order to wrongly trigger the recognition.

Our benchmark is based on a few canonical examples, declined in different binaries each, once compiled with different compilers and options. Those examples are based on the following patterns:

- Basic switch: 1 range of consecutive cases
- Double switch: 2 ranges of consecutive cases
- Mixed switch: 2 ranges of consecutive cases separated by spare values.
- Switch with almost contiguous cases: 1 range of cases minus one or two values

The following table represent a summary of our results: the first three columns describe the binary tested and the last two indicate if IDA recognize a switch statement using a jumptable and build the whole CFG, with or without our plugin. Some combinations of options and compilers generate similar results, so we only displayed the non-duplicated ones. We can notice the plugin obtains pretty good results.

<table>
<thead>
<tr>
<th>Type</th>
<th>Compiler</th>
<th>Option</th>
<th>IDA vanilla</th>
<th>IDA plugin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>GCC</td>
<td>O0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Basic</td>
<td>Clang</td>
<td>O0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Basic</td>
<td>GCC</td>
<td>O3</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Basic</td>
<td>Clang</td>
<td>O0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Mixed</td>
<td>Clang</td>
<td>O0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Mixed</td>
<td>Clang</td>
<td>O3</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Double</td>
<td>Clang</td>
<td>O0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Double</td>
<td>Clang</td>
<td>O3</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Non-contig.</td>
<td>Clang</td>
<td>O0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Non-contig.</td>
<td>GCC</td>
<td>O0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Non-contig.</td>
<td>GCC</td>
<td>O3</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

Regarding the false-positive tests, we first wrote a sample code which aimed to create an array in `.rodata` segment, then we progressively added some code to this example in order to come closer to the structure of a switch statement. Here is the last example (ie the closer to a switch statement):

```c
int main (int argc, char* argv[]){
    // We create a table in .rodata
    static const void const * tab[10] = {
        &zero, &one, &two, &three, &four, &five, &six, &seven, &eight, &nine};
    goto *variable;
}
```

![FIGURE 10: NegaTest5.c](image)

This code generate an assembly code structurally equivalent to a switch statement, that is why it is the only "false example" that triggers the switch recognition. The other tests are not wrongly recognized by our plugin.

You will find a set of the binaries we used on [https://sites.google.com/site/switchrecognition/resources/test-benchmark](https://sites.google.com/site/switchrecognition/resources/test-benchmark).
6. CONCLUSION

Throughout this paper we have draw attention to the limits of the binary analyzers and we found in which circumstances common compilers generate dynamic jump in order to translate a switch statement into binary code. Our plugin represents a proof of concept: with a syntactic approach, we obtained valuable results regarding the CFG reconstruction in the presence of a switch statement. Finally, even if our plugin handles all the examples we found, its analysis could be a bit more improved: it could for instance recognize the use of several jumptables that represent a single switch. At a visual level, adding edges between the blocks can simplify the navigation and the view of the CFG.

7. ACKNOWLEDGMENTS

We would like to thank to our referees, Marie-Laure POTET and Josselin FEIST, for the advices, for the constructive feedback on our work and for helping us to improve the content of this paper.

We would also like to express our special thanks to Karim HOSEN, who helped us to understand how the IDA Python API works.

8. REFERENCES